Expression et composition des motifs de conception avec les aspects

Simon Denier

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX

Année 2007

N° attribué par la bibliothèque

Expression et composition des motifs de conception avec les aspects

---

THÈSE DE DOCTORAT

Discipline : informatique

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Simon DENIER

Le 9 juillet 2007 à l'École des Mines de Nantes

Devant le jury ci-dessous

Président	:	Isabelle Borne, Professeur	Université de Bretagne Sud
Rapporteurs	:	Lionel Seinturier, Professeur	Université de Lille 1
		Mikal Ziane, Maître de conférences HDR	Université de Paris 5
Examinateurs	:	Pierre Cointe, Professeur	École des Mines de Nantes
		Mourad Oussalah, Professeur	Université de Nantes

Directeur de thèse : Pierre Cointe
DÉPARTEMENT INFORMATIQUE, ÉCOLE DES MINES DE NANTES
La Chantrerie – 4, rue Alfred Kastler – 44307 Nantes Cedex 3, France

N° ED 366-306

EXPRESSION ET COMPOSITION DES MOTIFS DE CONCEPTION AVEC LES ASPECTS

Expression and Composition of Design Motifs with Aspects

Simon Denier

favet neptunus eunti

Université de Nantes

Simon Denier

Expression et composition des motifs de conception avec les aspects

Résumé

Les patrons de conception répertorient les bonnes pratiques de la programmation par objets. Les solutions des patrons, appelées motifs, apparaissent avec une densité croissante dans les bibliothèques et cadriciels. Les effets de cette densité sur la modularité, l'adaptation et la réutilisation des programmes sont mal connus. Or la dispersion et le mélange du code lié à l'implémentation des motifs rendent difficile l'étude de ces effets. La programmation par aspects est une technique nouvelle dédiée au traitement de ces deux symptômes. En modularisant les motifs dans des aspects, nous pouvons analyser de manière plus fine les problèmes d'implémentation et de composition des motifs liés à leur densité.

Cette thèse aborde les problèmes de la densité, de l'implémentation et de la composition des motifs avec AspectJ, une extension de Java pour les aspects. À partir du cas concret du cadriciel JHotDraw, nous montrons qu'une forte densité est un facteur de risque sur la modularité et l'adaptation d'un programme objet. Nous présentons la transformation des motifs à l'aide des aspects et nous décrivons les idiomes d'AspectJ supportant leur modularisation. Nous examinons la modularité et la réutilisation des compositions de motifs définies avec les aspects. Nous proposons la résolution des interactions entre motifs à l'aide du langage de coupe des aspects. Enfin nous développons une méthode de programmation avec AspectJ basée sur l'usage conjoint des classes et des aspects. Ces travaux nous permettent de conclure sur l'intérêt des aspects comme moyen d'étude et de traitement de la densité des motifs. Ils ouvrent également des pistes pour l'amélioration des langages d'aspects.

Table des matières

Remerciements

Je remercie Pierre Cointe pour avoir accepté d'être mon directeur de thèse, il y a quatre ans de cela, et m'avoir laissé trouver ma propre voie dans la recherche. Je ne savais pas dans quelle direction je m'engageais alors, il y a eu de nombreux chemins de traverse entre temps, mais la route est là !

Je remercie également les membres de mon jury, et en particulier les rapporteurs, pour le beau diplôme encadré au mur bien sûr, mais surtout pour leur regard critique qui m'a aidé à clarifier mes propos et à dégager mes perspectives d'après-thèse.

Didier, Florian, Jacques, Luc, Marc, Rémi, Robin, Yann-Gaël... Vous m'avez aidé dans mes réflexions et supporté (parfois de façon bien inconsciente) dans mes moments de doute. Je regrette seulement de ne pas avoir pris plus de temps pour discuter et travailler avec vous.

Enfin, j'ai pu apprécier, à de nombreuses reprises, l'ambiance et le travail fournis par les membres du département informatique. Je leur exprime ma reconnaissance pour toutes ces expériences.

Je ne remercie pas mes trois Euménides : Procrastination, Dilettantisme et Internet.

Conventions typographiques

- Les mots en anglais, titres et citations sont en *emphase*.
- Les noms des patrons et des motifs sont en Petites Capitales.
- Les noms des classes et le code source en général sont dans une police à `chasse fixe`.
- Les entrées du glossaire sont en *emphase** suivi d'un astérisque.

Extended Abstract

This thesis explores some new cross-fertilizations between aspect-oriented programming and design patterns in object-oriented programming. Both themes have been actively researched in recent years, while trying to improve modularity, adaptation and reuse in object-oriented languages. Both can also be seen as approaches to tackle scalability problems with objects.

In particular, we focus on the high density of design motifs (solutions of design patterns), which appears whenever a program grows up and becomes mature in its conception. Our intuition is that density of design motifs is a vector for scalability in object-oriented programming - and consequently a way to study scalability on objects. We develop this problematic according to two topics: study of motifs density in object-based programs and implementation of motifs by aspects to better control the effects of a high density.

Linking Patterns and Aspects Problematics

Our first topic focus on studying of motifs density and its effects on object-based implementation. Intuitively, we define density as a ratio between a number of motifs (more exactly occurrences of these motifs) and a number of classes. Indeed, a motif generally involves several classes and a class can be involved in several motifs. For one class set, the larger the number of motifs is, the higher the density is.

But consequences of a high density on implementation are little-known, because motifs tend to disappear in the code. Motif implementation can be scattered across several classes. It can also be tangled with other concerns of one class. These symptoms point to the lack of modularity of motifs implementations.

Thus, our second topic is motifs implementation by aspects. Aspect-oriented programming aims at modularizing concerns hindered by code scattering and tangling with traditional techniques. Motifs whose object-based implementation shows these symptoms can be improved by the transformation. This also opens the way to motifs composition by aspects. In particular, we address the concern of implementation and modularity of these compositions.

Manuscript Outline

This manuscript explains the problematic of motifs density and aspects in a didactic way. In the first part we describe the state of the art in design motifs and aspect-oriented programming. Chapter 2 reviews implementation of design motifs in object-oriented languages. Chapter 3 presents aspect-oriented programming as well as previous work considering modularization of motifs by aspects.

The second part develops our four chapters of contributions. Chapter 4 inspects motifs density, its interpretation and its impact on object-based implementation. Chapter 5 demonstrates the transformation (or “aspectization”) of motifs by aspects – in particular new solutions possible with aspects and idioms suitable for the AspectJ language. Chapter 6 explores composition

of motifs as well as implementation of composition with aspects. Chapter 7 describes a programming method with AspectJ based on the complementarity of classes and aspects.

Chapter 8 concludes and discusses future work.

Contributions

The following sections sum up the contributions of our four central chapters.

Case Study on Motifs Density in JHotDraw – Chapter 4

We study the design and implementation of the JHotDraw framework to illustrate the abstraction power provided by motifs but also their impact on implementation in a high density setting [Denier et Cointe, 2006b].

Interpretation of motifs density. A high density of design motifs means that many concerns of the framework are supported by these motifs. We discriminate two uses: motifs ensuring an internal collaboration or structure within the framework and motifs allowing framework extension. We show how motifs can explain the implementation in a synthetic way. We then present two simple metrics for density to evaluate code maturity and to identify essential classes, called entities, for extension and collaborations. These descriptions and metrics produce a more abstract vision of the implementation while remaining connected to the code.

Impact of motifs on object-based implementation. We test the impact of new motifs on the framework implementation. Modifications are invasive because of the lack of modularity of object-based solutions. We also notice a crosscutting impact, which we link to the high density: choices pertaining to a motif in one class can alter the implementation of another motif in one other class. We observe that a high density provides opportunities for simplification and reuse of motifs code. But these opportunities imply a tradeoff between motifs specifications and weaken their implementations. This case study thus illustrates the effects of a high density on motifs implementation.

Implementation of Design Motifs with AspectJ – Chapter 5

Motifs implementation by aspects aims at a better modularization [Denier et Cointe, 2006a]. This approach is based on the transformation of object-based solutions by aspects mechanisms. Two types of solutions come up: idiomatic solutions and aspect-based solutions.

Idiomatic solutions and aspect-based solutions. The idiomatic approach is a partial transformation of the object-based solution completed by modularization of other elements in an aspect: this approach tends to reproduce the initial object-based solution with idioms. We describe the idioms suitable for AspectJ. The aspect-based approach tackles the problem addressed by the pattern with a new solution, different from the object-based one. We present two successful solutions, Decorator and Visitor. These solutions have different properties from their object-based counterparts in term of implementation (simplicity, genericity) and behavior (use, instantiation and recursivity). Some behaviours can be emulated with other AspectJ idioms, which we demonstrate. AspectJ idioms point to a connection concern between aspects

and objects: they deal with defining objects relation within a motif, or linking the state or the scope of an aspect to an object.

Support of *GoF* motifs with AspectJ. We present an interpretation guide of motifs implementations realized by [Hannemann et Kiczales, 2002] for the *GoF* catalog [Gamma *et al.*, 1994]. This guide compares the reusability of the new solutions with the AspectJ mechanisms used. It shows that several motifs using AspectJ idioms are reusable. The non-reusable motifs get help from the AspectJ support for structural extensions using intertypes declarations. In general, modularization of motifs by aspects is effective. However, the outcome of aspectization varies significantly from one motif to another. Most motifs can be completely defined in aspects. But some get only a partial support from aspects: the programmer must complement the implementation with pieces of the object-based solution.

Composition of Design Motifs with AspectJ – Chapter 6

The modular implementation of motifs with aspects enables studying motifs composition in the code [Denier et Cointe, 2006a]. We survey the modularity of motifs compositions. In particular, we are interested in aspects mechanisms for the resolution of interactions and conflicts between motifs, which are both potential negative effects arising with a high density of motifs.

Modularity and modality of composition. Depending on the language and the program specifications, the programmer implements a motifs composition with a tradeoff between modularization and invasion of code. We propose the concept of modality to announce an expected degree of modularity for such a composition. We demonstrate the tradeoffs and properties attached to two different modalities for the composition of the Composite and Visitor motifs. This example also demonstrates that making a reusable composition of motifs is possible.

Mechanisms for motifs composition and resolution of interactions. An interaction between motifs signals an in-depth alteration of behaviour and thus of implementation following the composition. We describe AspectJ mechanisms helping to solve some structural and behavioral interactions between motifs. Intertypes declarations support structural composition in AspectJ. We compare and underline the proximity between intertypes declarations and traits, a new mechanism for reuse [Schärli *et al.*, 2003]. However, the traits model is more flexible with respect to conflicts resolution, due to its fine-grained operators [Denier, 2005]. We suggest two schemes for the static typing of traits, in the perspective of their integration with AspectJ. Crosscut language can help with behavioral composition. We demonstrate the resolution of interactions in a motifs composition using control flow crosscuts. This example illustrates how aspects allow to cleanly solve these interactions out of classes context.

Programming Method with AspectJ – Chapter 7

The implementation and composition of design motifs with AspectJ use both objects and aspects. We develop a programming method for a combined and complementary use of classes and aspects. This method is especially useful, but not restricted, for refactoring of object-based programs with aspects. It explains criteria to resolve benefits vs drawbacks of AspectJ modularization. We stress two points (summarized in the table below) to discriminate between classes

and aspects. We distinguish modules "types" (class, trait, aspect) according to different roles: classes for instance creation and inheritance, traits (or structural aspects) for reuse of code, behavioral aspects for some external collaborations between classes. We also distinguish according to the adaptation offered by these modules: classes (and by extension traits, or structural aspects) support subtyping polymorphism (redefinition), while AspectJ behavioral aspects support ad-hoc polymorphism (overloading). Through this method, we advise a balanced approach of motifs aspectization, rather than a systematic transformation.

Domain	Module	Polymorphism
Structural	class/trait	inheritance
Behavioral	aspect	ad-hoc

Conclusion and Future Work

Our work supports the idea that density of design motifs is a vector for scalability in object-oriented programming. A high density of motifs facilitates code abstraction and extension but weakens its object-based implementation and modularity. Studying motifs density must be continued in order to improve its interpretation as well as to describe its consequences.

Our work also validates the process of implementing motifs with aspects. As shown by Hannemann *et al.*, modularization of motifs by aspects is effective. It allows reasoning on the implementation and composition of motifs. The aspects free the motif from the class. We think that this research must be developed in conjunction with further studies on density. In addition, this approach should be performed and compared with other aspect-oriented languages (Caesar [Mezini et Ostermann, 2003], Reflex [Tanter, 2004]) and programming techniques (traits and classboxes [Minjat *et al.*, 2005]).

Finally our work focuses on good practices for aspect-oriented programming. Results with AspectJ are uneven in term of code simplicity, reuse, and motifs properties. AspectJ idioms underline some shortcomings of the language. For this reason, we advise a balanced use of AspectJ for practical purposes. But these problems also hint at further research towards a better integration of objects and aspects.

Chapitre 1

Introduction

Sommaire

Préambule : complexité du développement logiciel et langages de programmation

Face à la complexité croissante des systèmes informatiques, encadrée par les méthodes du génie logiciel, la programmation demeure l'activité pivot du cycle de développement logiciel. Ce cycle décompose le développement en plusieurs phases autour de l'implémentation. Les phases préparatoires visent la spécification et la conception du système ; les phases postérieures incluent le test et la maintenance. Ce découpage en tâches se reflète aussi dans la décomposition du système en préoccupations : chaque préoccupation est isolée pour en résoudre les difficultés avant de les recomposer. La séparation des préoccupations est au cœur des phases de conception, d'implémentation et de maintenance du programme.

Mais sans usage amont des méthodes formelles du génie logiciel, le code reste le produit final, rendant compte seul de la qualité du projet. D'une certaine façon, le code définit ses propres spécifications, conséquences des contraintes et accidents du développement, en adéquation (en grande partie) avec celles définies pour le projet. Cet état de fait illustre le côté encore artisanal de la programmation : le résultat dépend de l'expérience de chaque programmeur et des outils qu'il utilise.

Le langage de programmation est à ce titre le premier « outil » du programmeur. Le langage doit assurer deux fonctions essentielles : capturer les abstractions du programmeur (de la conception) et garantir une traduction cohérente vers la machine (grâce à une sémantique). Enfin une troisième fonction est importante : le langage doit permettre le passage à l'échelle

du développement, c'est-à-dire offrir les mêmes facilités et garanties pour un grand programme que pour un petit programme. La recherche de la modularité dans les langages vise directement le passage à l'échelle à travers la séparation des préoccupations dans le code. En pratique, la programmation reste une activité coûteuse en terme de temps et d'expertise : les mécanismes des langages pour la réutilisation et l'adaptation du code supportent aussi le passage à l'échelle.

1.1 Le passage à l'échelle en programmation par objets

La programmation par objets s'est imposée progressivement comme un paradigme important dans le monde du développement logiciel. Ce succès s'explique par la facilité avec laquelle le concept d'objet semble capturer la réalité : l'objet est une abstraction générale. Le concept a été décliné dans des langages purement objets (Smalltalk), mixtes (Java, C++) et multi-paradigmes (OCaml, Scala) : il a été adapté aux différentes sémantiques de ces langages. Les langages à objets ont donc des bases abstraites et sémantiques bien connues. Par ailleurs ils se sont confrontés au passage à l'échelle dans des projets informatiques d'envergure, tels les systèmes d'exploitation, les systèmes d'information des entreprises, ou encore les intergiciels pour la programmation distribuée – domaines où de nombreux travaux de recherche indiquent que des problèmes persistent.

Or les langages à objets, pour la plupart, sont conçus autour du concept de classe. La classe détermine plusieurs mécanismes des langages à objets liés en particulier au passage à l'échelle. Elle définit l'encapsulation propre à un objet ; elle supporte la réutilisation par héritage ; elle permet l'adaptation par polymorphisme d'héritage. Nous exposons ci-dessous divers travaux de recherche qui ont développé ces pistes en proposant des solutions complémentaires ou alternatives.

1.1.1 Réutilisation : cadriciels et patrons de conception

L'exploitation de l'héritage comme mécanisme central de réutilisation mène à la construction de cadriciels. Les *cadriciels** sont des squelettes réutilisables de programmes. Ils sont conçus autour d'un ensemble de classes abstraites que le programmeur doit spécialiser pour son application. Par ailleurs le flot de contrôle dans un cadriciel est souvent inversé : les classes du cadriciel font appel, grâce au polymorphisme, aux classes de l'application. De cette façon le programmeur peut se concentrer sur la seule définition des classes spécifiques à son application. L'approche des cadriciels vise donc à créer des programmes réutilisables pour des domaines d'application particuliers.

Cependant le développement de nombreux programmes objets et en particulier des cadriciels montre que l'art est délicat. Le développement d'un cadriciel nécessite de nombreuses itérations pour arriver au degré d'abstraction nécessaire à sa réutilisation. Le code d'un cadriciel forme un bloc de classes fortement corrélées : il est difficile de ne réutiliser qu'une partie du cadriciel. Il est aussi difficile de généraliser un cadriciel pour le réutiliser dans un autre domaine d'application.

Pourtant l'expérience acquise en programmation par objets montre des solutions récurrentes à défaut d'être réutilisables. Les *patrons de conception** capturent cette expérience en formalisant le problème et sa solution (que nous appelons *motif de conception**) dans le langage naturel [Gamma *et al.*, 1994]. Les patrons de conception sont assimilés aux bonnes pratiques de la programmation par objets. Ainsi une implémentation mature d'un programme présente une *forte densité* de motifs de conception [Gamma et Beck, 2002], suggérant que le fonctionnement d'un

programme objet est supporté par ces motifs. Ceci indique que ces recettes éprouvées sont nécessaires pour faire passer l'intuition de l'objet à l'échelle mais échappent en partie aux mécanismes classiques de réutilisation que sont l'agrégation et l'héritage.

1.1.2 Adaptation et modularité : langage réflexif et extension des langages

La recherche d'une meilleure adaptation s'est illustrée avec le développement de la réflexion [Smith, 1984], qui vise l'ouverture et l'extension des langages de programmation. Un langage réflexif représente et manipule sa propre structure et son comportement : il peut donc être étendu avec de nouvelles abstractions et une sémantique adaptée à un problème particulier. La réflexion repose sur la distinction entre deux niveaux de programmation en connection causale[1] : un niveau de base (ou applicatif) et un méta-niveau. Les protocoles de métaobjets exploitent cette distinction pour mettre l'accent sur la séparation des préoccupations entre le programme, qui doit implémenter le métier de l'application, et le langage de programmation, qui doit fournir les moyens de l'implémentation [Kiczales *et al.*, 1991]. Ainsi un cadriciel est un niveau applicatif spécialisé pour un domaine particulier ; un langage réflexif peut définir des abstractions facilitant l'implémentation de ce cadriciel. La réflexion ouvre la voie à l'adaptation du langage aux problèmes d'implémentation posés par ses propres mécanismes.

Mais le niveau d'abstraction, le coût à l'exécution et la nature des modifications permise par la réflexion rendent son usage délicat dans le développement logiciel à grande échelle. Des extensions plus simples et moins coûteuses sont donc privilégiées. Parmi les langages à classes, la programmation par sujets [Harrison et Ossher, 1993], la programmation adaptive [Lieberherr, 1996] et la programmation par aspects [Kiczales *et al.*, 1997] explorent des pistes alternatives visant à corriger les défauts d'adaptation et de modularité des programmes à objets.

La programmation par aspects est historiquement une héritière des travaux sur la réflexion et les protocoles de métaobjets (MO). Elle se base, comme la réflexion, sur une représentation du programme avec les éléments du langage. Mais si la réflexion et les protocoles MO proposent une solution générale à la séparation des préoccupations par l'ouverture du langage, la programmation par aspects s'intéresse au problème particulier de la modularité des préoccupations transversales. Suivant cette problématique, la décomposition d'un système informatique en préoccupations ne cadre pas toujours avec un système hiérarchique de classes : certaines préoccupations sont « forcées » dans les classes existantes. En terme de symptômes, les implémentations de ces préoccupations sont dispersées et mélangées avec les autres préoccupations. La programmation par aspects est une technique visant une bonne modularisation de ces préoccupations, en complément des mécanismes préexistant dans les langages classiques [Filman *et al.*, 2005].

1.2 Problématiques croisées des motifs et des aspects

Ces approches de la réutilisation, de l'adaptation et de la modularité, dans les langages à classes, forment les points de départ de notre thèse. En particulier notre intuition est que la densité des motifs de conception est un vecteur du passage à l'échelle en programmation par objets – et par conséquent un moyen d'étude de ce passage et de ses effets sur les objets. Nous développons cette problématique suivant deux thèmes de travail : l'étude de la densité des motifs

[1] Les deux niveaux sont corrélés, de façon que toute modification dans un des niveaux se répercute sur l'autre.

dans les programmes objets et l'implémentation des motifs par les aspects pour mieux maîtriser les effets d'une forte densité.

1.2.1 Étude de la densité des motifs de conception

Notre premier thème de travail est l'étude de la densité des motifs et de ses effets sur l'implémentation. Intuitivement nous définissons la densité comme le rapport entre un nombre de motifs (plus exactement d'*occurrences** de ces motifs) et un nombre de classes. En effet un motif implique généralement plusieurs classes et une classe peut se trouver impliquée dans plusieurs motifs. Pour un même ensemble de classes, plus le nombre de motifs impliqués sera grand, plus la densité sera forte.

Les effets d'une forte densité sur l'implémentation d'un programme sont mal connus. L'objectif d'un motif est de permettre ou d'améliorer une caractéristique du programme, comme ses collaborations entre classes ou ses extensions modulaires. Mais l'implémentation du motif crée des dépendances entre différentes classes et demande de modifier leur code au détriment de leur modularité. En ce sens, les motifs entraînent des micro-violations de modularité, à l'échelle d'une classe, dans le but d'obtenir une meilleure macro-modularité, à l'échelle d'un ensemble de classes. Une forte densité de motifs peut donc avoir un impact sur l'implémentation à la fois positif et négatif : positif sur l'extension immédiate de l'application (mais aussi en termes d'abstraction et de documentation pour la conception et la maintenance) ; négatif en cas d'évolution imprévue car la modularité de chaque classe est remise en cause.

Ces conséquences restent difficiles à évaluer en pratique car les implémentations des motifs disparaissent dans le code. D'une part le code lié à un motif peut être dispersé dans plusieurs classes. D'autre part ce code se trouve souvent mélangé avec les autres préoccupations de la classe. Ces symptômes ne sont ni spécifiques aux motifs, ni n'affectent tous les motifs. Mais ils indiquent la non-modularité de plusieurs de leurs implémentations.

1.2.2 Implémentation des motifs par les aspects

Notre deuxième thème de travail suit naturellement avec l'implémentation des motifs par les aspects. En effet la programmation par aspects vise la modularisation des préoccupations en traitant les symptômes de dispersion et de mélange du code. Les motifs dont l'implémentation objet manifeste ces symptômes peuvent donc bénéficier de ce traitement. Cette étape ouvre aussi la voie à la composition des motifs à l'aide des aspects. En particulier nous abordons la question du point de vue de l'implémentation et de la modularité de ces compositions. Cette démarche doit faciliter à terme l'implémentation d'applications à forte densité de motifs.

Cependant la modularisation par les aspects implique l'utilisation des nouveaux mécanismes de programmation associés aux aspects. Les motifs, basés initialement sur les mécanismes des objets, doivent donc être partiellement transformés. Les solutions des motifs aspectisés peuvent alors avoir des propriétés différentes des motifs originaux (tout en réalisant l'intention du patron). Ces nouvelles propriétés influent sur l'utilisation de ces nouvelles solutions dans les programmes.

Ce dernier point soulève la question de la complémentarité des objets et des aspects. En effet, si les motifs sont des bonnes pratiques de l'objet, la question se pose quant aux bonnes pratiques des aspects et à l'usage conjoint avec les objets. En filigrane apparaît le problème de l'intégration des objets et des aspects dans les langages de programmation.

1.3 Plan de lecture et contributions

Ce manuscrit développe la problématique de la densité des motifs et des aspects suivant une progression didactique. Dans la première partie nous dressons un état de l'art des motifs de conception et de la programmation par aspects. Le chapitre 2 réalise un bilan de l'implémentation des motifs de conception dans les langages à objets. Le chapitre 3 présente la programmation par aspects ainsi que les travaux relevant de la modularisation des motifs par les aspects.

La seconde partie développe nos contributions sur quatre chapitres. Elle commence par une brève synthèse de l'état de l'art : cette synthèse reprend les éléments servant de points de départ à nos travaux. Le chapitre 4 explore la densité des motifs, son interprétation et son impact sur l'implémentation des programmes objets. Le chapitre 5 présente la transformation (ou « aspectisation ») des motifs par les aspects, en particulier les solutions originales des aspects et les idiomes propres à AspectJ. Le chapitre 6 examine la composition de motifs et aborde l'implémentation de la composition avec les aspects. Le chapitre 7 expose une méthode de programmation avec AspectJ basée sur la complémentarité des classes et des aspects – elle est appliquée aux cas d'étude présentés dans le chapitre 4.

Le chapitre 8 regroupe (modularise) ces contributions dans un bilan et discute des travaux futurs autour de la densité des motifs et de la programmation par aspects.

Cadre de l'étude. Nous nous intéressons uniquement aux motifs de conception de la programmation par objets et en particulier aux motifs du *GoF*[2] [Gamma *et al.*, 1994]. Dans la suite de ce manuscrit, le terme motif désigne toujours un motif de conception en programmation par objets. Nous limitons par ailleurs cette étude aux langages Java et AspectJ, en raison de la maturité industrielle de ce dernier et des études préexistantes sur les motifs [Hannemann et Kiczales, 2002].

Plans et niveaux de lecture alternatifs. Deux niveaux de lecture cohabitent dans cette thèse et s'adressent à deux publics différents : le programmeur et le concepteur de langage.

Le programmeur sera intéressé par la description des usages et des solutions des motifs avec les aspects, assurée par la progression didactique des chapitres. Les chapitres 2 et 4 concernent exclusivement les motifs de conception dans les langages à objets : ils peuvent donc servir comme entrée en matière sur les motifs et la densité pour une personne ne connaissant pas ou peu les aspects.

Le concepteur de langage sera intéressé par les problèmes d'implémentations et l'impact des motifs dans les programmes à objets (deuxième moitié du chapitre 4) mais aussi par l'intégration des objets et des aspects et les lacunes d'AspectJ exposés dans les chapitres 5 et 6 – ainsi que, dans une moindre mesure, par la méthode de programmation présentée dans le chapitre 7.

Enfin la figure 1.1 (page 9) dresse un panorama des principaux motifs explorés dans ce manuscrit et de la progression logique de notre démarche à travers ces exemples.

Les sections suivantes détaillent le contenu des quatres chapitres de la partie Contributions.

[2] *Gang of Four* : surnom donné au groupe des quatres auteurs et par éponymie, surnom du livre. Nous utiliserons cette acronyme par la suite.

1.3.1 Étude de cas sur la densité des motifs dans JHotDraw

Nous étudions la conception et l'implémentation du cadriciel JHotDraw[3], dont l'usage des motifs est documenté dans le code. Cette étude illustre le pouvoir d'abstraction des motifs mais aussi leur impact sur l'implémentation dans le cadre d'une forte densité [Denier et Cointe, 2006b].

1.3.1.1 Interprétation de la densité des motifs

Une forte densité de motifs de conception indique que de nombreuses préoccupations du cadriciel sont supportées par ces motifs. Nous distinguons deux usages caractéristiques : les motifs assurant une collaboration ou une structure interne au cadriciel et les motifs permettant une extension du cadriciel. Nous montrons comment les motifs présents dans le code expliquent de façon synthétique l'implémentation.

Nous présentons ensuite deux métriques simples de densité destinées à : évaluer la maturité du code (d'après la part assurée par les motifs) ; identifier les classes essentielles pour l'extension et les collaborations, appelées *entités** .

Ces descriptions et métriques permettent de produire une vision plus abstraite de l'implémentation tout en restant connectées au code.

1.3.1.2 Impact des motifs sur l'implémentation objet

En rajoutant au cadriciel de nouvelles extensions, nous observons l'impact de nouveaux motifs sur l'implémentation du cadriciel. Les modifications apportées sont invasives d'abord en raison de la non-modularité des solutions objets. Mais, dans le cadre d'une forte densité, nous constatons aussi un impact transversal : les choix concernant un motif défini dans une première classe affectent l'implémentation d'un autre motif dans une seconde classe.

Nous observons qu'une forte densité est source d'opportunités pour la simplification et la réutilisation du code des motifs. Mais ces opportunités impliquent un compromis entre les motifs concernés et fragilisent les implémentations.

Cette étude de cas concrète illustre donc les effets de l'implémentation des motifs dans le cadre d'une forte densité.

1.3.2 Implémentation des motifs de conception avec AspectJ

Le but principal de l'implémentation des motifs avec les aspects est leur modularisation [Denier et Cointe, 2006a]. Cette démarche repose sur la transformation des solutions objets par les mécanismes aspects. Deux types de solutions se dégagent suivant le degré de transformation atteint : les solutions idiomatiques et les solutions aspects.

1.3.2.1 Solutions idiomatiques et solutions aspects

L'approche idiomatique transforme partiellement la solution et modularise les autres éléments du motif dans un aspect : cette approche tend à reproduire dans l'aspect la solution objet initiale. Nous décrivons cette approche avec les idiomes propres à AspectJ.

L'approche par aspects capture le problème visé par le patron dans une nouvelle solution, différente de la solution objet initiale. Nous présentons les deux cas les plus aboutis, DECORATOR

[3] Accessible à l'adresse `http://www.jhotdraw.org/`.

et Visitor. Ces solutions ont des propriétés différentes de leurs homologues objets en terme d'implémentation (simplicité, généricité) mais aussi de comportement (utilisation, instantiation et récursivité). Certaines propriétés de comportement peuvent être émulées avec d'autres idiomes d'AspectJ que nous avons développés.

D'une manière générale, ces idiomes AspectJ concernent la liaison des aspects aux objets : il s'agit de définir une relation particulière entre objets ou encore d'associer l'état ou la portée d'un aspect à un objet.

1.3.2.2 Étendue du support des motifs du *GoF* avec AspectJ

Pour évaluer le résultat de l'aspectisation des motifs avec AspectJ, nous présentons une grille de lecture des implémentations des motifs du *GoF* réalisées par Hannemann *et al.*. Cette grille place le degré de réutilisation des solutions des motifs en regard des mécanismes AspectJ utilisés. Elle montre que plusieurs des motifs utilisant les idiomes AspectJ sont réutilisables. Les motifs non réutilisables peuvent bénéficier du support d'AspectJ pour les extensions structurelles à l'aide du mécanisme des déclarations intertypes. D'une manière générale, la modularisation des motifs par les aspects est effective.

Cependant le résultat de l'aspectisation par AspectJ change d'un motif à l'autre. La plupart des motifs peuvent être complètement définis dans des aspects. Mais certains motifs, même réutilisables, ne sont qu'en partie supportés par les aspects : le programmeur doit compléter la définition du motif en s'inspirant de la solution objet originale.

1.3.3 Composition des motifs de conception avec AspectJ

L'implémentation modulaire des motifs avec les aspects permet d'explorer dans le code la composition des motifs [Denier et Cointe, 2006a]. Nous examinons la modularité des compositions de motifs. Nous nous intéressons en particulier aux mécanismes des aspects permettant la résolution des interactions et des conflits entre motifs, effets négatifs liés à une forte densité.

1.3.3.1 Modularité et modalité de composition

En fonction du langage de programmation et des spécifications du code, le programmeur peut aboutir à un compromis entre modularisation et invasion de code pour implémenter une composition. Nous proposons le concept de *modalité** pour désigner et discuter le degré de modularité attendu (soit aussi, le degré d'imbrication autorisé) dans la composition de deux motifs.

Nous illustrons avec les motifs Composite et Visitor les compromis et les propriétés attachés à deux modalités de composition différentes. Cet exemple montre aussi que la définition d'une composition réutilisable est possible.

1.3.3.2 Mécanismes de composition et de résolution d'interactions entre motifs

Une interaction entre motifs indique que la composition change en profondeur le comportement et donc l'implémentation d'au moins un des motifs. Nous décrivons les mécanismes d'AspectJ pour résoudre les interactions structurelles et comportementales entre motifs.

La composition structurelle repose sur les déclarations intertypes d'AspectJ. Nous comparons et soulignons la proximité entre les déclarations intertypes (aspects structurels) et les traits, un

nouveau mécanisme pour la réutilisation [Schärli *et al.*, 2003]. Cependant le modèle des traits offre une plus grande finesse dans la résolution des conflits grâce à ses opérateurs [Denier, 2005]. En vue d'une intégration de ces opérateurs à AspectJ, nous concluons sur les solutions que nous préconisons pour le typage statique des traits.

La composition comportementale peut s'opérer avec le langage de coupe des aspects. Nous montrons la résolution d'interactions dans une composition de motifs à l'aide des coupes sur le flot de contrôle. Cet exemple illustre l'affranchissement nécessaire – et permis par les aspects – pour résoudre ces interactions hors du contexte des classes.

1.3.4 Méthode de programmation avec AspectJ

L'implémentation et la composition des motifs avec le langage AspectJ utilise objets et aspects. Nous développons une méthode de programmation pour un usage conjoint et complémentaire des classes et des aspects.

Cette méthode s'applique en particulier à la restructuration des objets par les aspects. Elle propose plusieurs critères permettant de décider si la modularisation par AspectJ est intéressante. Nous mettons l'accent sur deux points (résumés dans le tableau ci-dessous) discriminant l'usage des classes et des aspects. Nous distinguons les catégories de module (classe, trait et aspect) suivant différents « rôles » : les classes pour la définition d'instances et l'héritage, les traits (ou aspects structurels) pour la réutilisation de code, les aspects comportementaux pour les collaborations externes aux classes. Nous distinguons aussi suivant le type d'adaptation offert par ces modules : les classes (et par extension les traits ou aspects structurels) supportent le polymorphisme d'héritage (redéfinition), les aspects comportementaux d'AspectJ supportent le polymorphisme ad hoc (surcharge).

Domaine	Module	Polymorphisme
Structurel	classe/trait	héritage
Comportemental	aspect	ad hoc

Nous appliquons cette méthode à JHotDraw et en particulier aux cas de densité présentés dans l'étude au chapitre 4. Nous préconisons à travers cette méthode une approche mesurée de l'aspectisation des motifs plutôt qu'une transformation systématique.

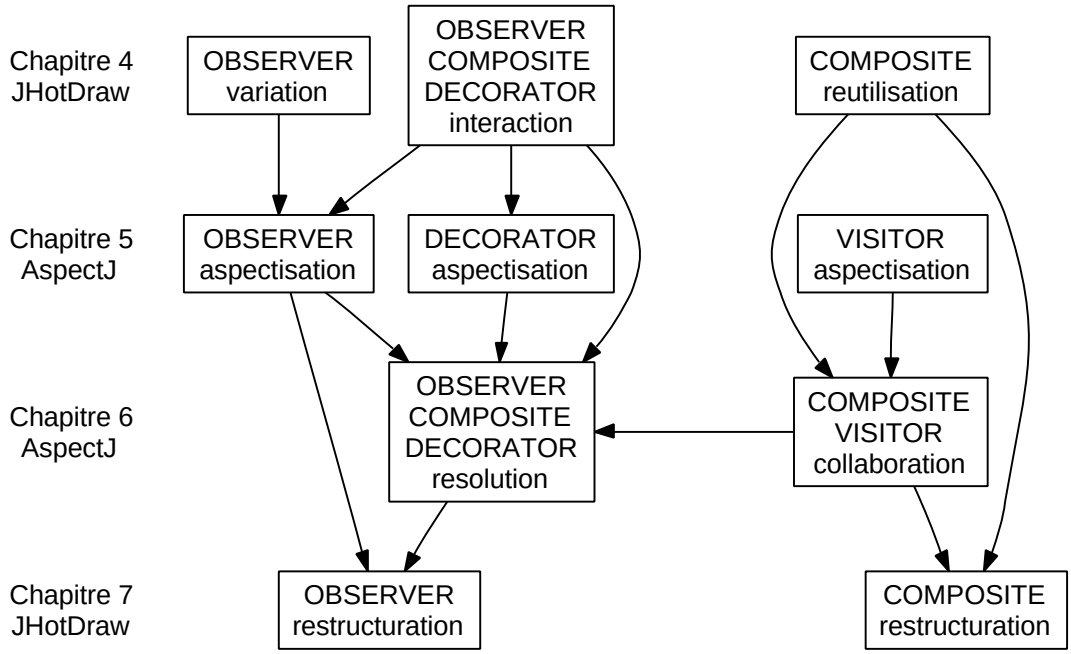
1.4 Conclusion

Nos travaux appuient l'idée que la densité des motifs de conception est un vecteur du passage à l'échelle en programmation par objets. Une forte densité de motifs facilite l'abstraction du code mais fragilise l'implémentation et la modularité du programme. L'étude de la densité doit être poursuivie pour en améliorer l'interprétation et en décrire les conséquences.

Nos travaux valident aussi l'approche visant l'implémentation des motifs avec les aspects. Comme indiqué par Hannemann *et al.*, la modularisation des motifs par les aspects est effective. Elle autorise le raisonnement sur l'implémentation et la composition des motifs. Les aspects permettent de s'affranchir de la classe comme module. Nous pensons que cette voie de recherche doit être développée en s'inspirant entre autre des études sur la densité. Il serait par ailleurs intéressant de conduire cette approche avec d'autres langages d'aspects (Caesar [Mezini et Ostermann, 2003], Reflex [Tanter, 2004]) et d'autres techniques de programmation (traits et *classboxes* [Minjat *et al.*, 2005]) à fin de comparaison.

Enfin notre thèse explore les bons usages de la programmation par aspects. Les résultats obtenus avec AspectJ sont inégaux en terme de simplicité d'implémentation, de réutilisation et de propriétés des motifs. Les idiomes AspectJ soulignent les lacunes du langage. C'est pourquoi nous préconisons en pratique une utilisation mesurée d'AspectJ. Mais ces défauts sont autant de nouvelles pistes de recherche pour une meilleure intégration des objets et des aspects.

Illustration 1.1 – Relations entre les principaux exemples développés dans ce manuscrit. Cette carte fait la synthèse de notre démarche à travers les motifs étudiés.



Partie I

État de l'art

Sommaire

Chapitre 2

Implémentation des motifs de conception

Sommaire

Ce chapitre fait le point sur les relations entre langages de programmation et motifs de conception. L'implémentation des motifs est un sujet difficile en raison de la disparité des problèmes et solutions abordées et de la relativité des patrons vis-à-vis des langages de programmation. Cependant un ensemble de problèmes communs à la plupart des patrons est identifié et fait référence à leur manque de modularité. Ce défaut nuit en particulier à l'étude de la composition des motifs. Les langages et les outils de programmation sont les deux domaines actifs dans la résolution de ces problèmes liés à la modularité. Les langages peuvent fournir de nouveaux mécanismes subsumant les patrons. Les outils intègrent les patrons au cycle de programmation.

2.1 Un court panorama des patrons de conception

L'idée des patrons, en tant que guides de différents aspects du développement logiciel, a émergé dans les années 80 avec la programmation par objets : celle-ci correspondait en particulier au besoin de partager et faire comprendre les principes des objets à travers l'expérience acquise. L'idée est directement inspirée, dans la forme et dans l'objectif, du concept original proposé par Christophe Alexander en architecture. Elle a été reprise puis expérimentée à travers plusieurs travaux.

Le livre *« Design Patterns: Elements of Reusable Object-Oriented Software »* par Gamma *et al.* [1994] marque la reconnaissance et la diffusion des patrons dans les milieux informatiques. Ce

livre en discute le principe, fait une démonstration de leur utilisation et présente un catalogue de vingt-trois patrons considérés comme standards.

Bien qu'il existe des patrons dédiés à différentes activités, le *GoF* n'aborde que les patrons de *conception*. L'expérience partagée par le *GoF* concerne deux activités universelles aux programmeurs, la conception et l'implémentation. Il tente aussi d'éclairer la relation des patrons aux langages et réciproquement.

2.1.1 Définitions

Nous nous intéressons dans cette thèse à la relation entre conception et implémentation et nous concentrons donc sur les seuls patrons de conception. En dehors de cette spécificité, la définition d'un patron de conception est similaire à celle de tout patron.

> Définition 2.1 – Patron de conception
>
> Un patron de conception décrit un problème récurrent en conception logicielle, propose une solution générique à ce problème, discute les compromis liés à cette solution et identifie l'ensemble par un nom.

La définition 2.1 distingue les quatre éléments constitutifs d'un patron : *nom, problème, solution, compromis (ou conséquences)*. Elle distingue aussi les propriétés caractéristiques d'un patron : la récurrence du problème et, à défaut de réutilisation, la généricité et les compromis de la solution.

L'implémentation de la solution générique du patron constitue une étape importante pour deux raisons : parce qu'elle résoud le problème bien sûr mais aussi parce qu'elle modifie le code. Ceci implique un coût de production immédiat ainsi qu'un coût de maintenance futur. C'est pourquoi nous nous intéressons plus spécifiquement dans la suite à cette solution et à l'impact qu'elle peut avoir sur le code. Suivant [Guéhéneuc, 2003], nous utilisons alors la terminologie suivante :

> Définition 2.2 – Motif de conception
>
> Un motif de conception est la solution générique au problème homonyme posé par le patron. Un motif décrit le squelette de cette solution ainsi que les choix et variations les plus courantes.

Cette distinction conceptuelle entre *patron de conception** et *motif de conception** se retrouve dans la terminologie : un patron *guide* le programmeur dans la résolution d'un problème ; un motif décrit la forme de la solution, répétée à chaque application du patron.

2.1.2 Utilisations

Suivant [Gamma *et al.*, introduction, sections 1.1, 1.6 et 6.1] et [Agerbo et Cornils, 1998], dans un objectif global de diffusion et formation aux principes objets, les patrons servent à :

1. capturer et partager l'expérience acquise en conception ;
2. créer un vocabulaire commun facilitant l'échange entre programmeurs.

Un patron se présente comme une discussion en langage naturel structurée par différentes rubriques. Le langage naturel permet d'exprimer les idées du patron de façon concise, quel que soit le niveau d'abstraction de ces idées, mais rend plus difficile la formalisation du patron. L'auteur du patron est cependant guidé (restreint) par les rubriques décrivant différents aspects du patron. Au sein d'un catalogue, les patrons respectent le même format des rubriques.

Comme décrit dans [Gamma *et al.*, sections 1.7, 1.8 et 6.1], il existe différents chemins pour lire les rubriques du patron. Dans le contexte d'un projet informatique, ces chemins correspondent à différentes phases du développement :

- conception : résolution des problèmes identifiés grâce aux patrons et choix des compromis [Gamma *et al.*, section 1.7] ;
- implémentation : adaptation au programme des motifs suggérés par les patrons [Gamma *et al.*, section 1.8] ;
- maintenance : documentation du code liée aux patrons [Agerbo et Cornils, 1998].

2.2 Difficultés dans l'implémentation des motifs de conception

La description des motifs en langage naturel facilite un exposé abstrait du principe et des variations courantes, mais est sujette à interprétation. Ce caractère informel mène à plusieurs conséquences en terme de disparité, d'implémentation et de composition des motifs. Nous exposons plusieurs travaux pertinents et les discutons.

2.2.1 Disparité des motifs de conception

Bien que les patrons de conception discutent de problèmes de même nature, le niveau d'abstraction et la complexité des motifs varie d'un patron à l'autre. Un motif tel que Template Method est une application directe de la spécialisation d'une méthode par héritage. Un motif tel que Memento 1) implique au moins trois classes et plusieurs collaborations entre elles ; 2) décrit de façon abstraite les activités engagées dans la création et la récupération de sauvegardes d'objets, sans détailler le mécanisme propre à cette sauvegarde.

Cette disparité est liée à l'usage du langage naturel, qui permet des descriptions concises mais parfois ambigues ou imprécises. Elle rend difficile l'évaluation du travail d'implémentation demandé par un motif ou la comparaison de deux motifs en terme d'implémentation. Plusieurs travaux, décrits ci-dessous, proposent des classifications afin de mieux organiser les catalogues des patrons, en particulier vis-à-vis du langage visé pour l'implémentation.

2.2.1.1 Relativité des motifs suivant les langages de programmation

L'utilisation par les motifs décrits dans le *GoF* de mécanismes objets basés sur les langages « standards » de l'époque (Smalltalk et C++) n'implique pas que ces motifs soient les solutions les plus adaptées à tout langage. Au contraire la présence d'un mécanisme particulier à un langage peut changer la facilité d'implémentation d'un motif, voire rendre le patron original « obsolète » [Gil et Lorenz, 1997]. [Gamma *et al.*, section 1.1] cite le typage dynamique en Smalltalk, facilitant l'écriture d'itérateurs génériques pour les collections, ou les multi-méthodes en CLOS, lesquelles rendent le patron Visitor moins utile.

Suivant ce point de vue, [Gamma *et al.*, 1994] avance l'idée qu'un langage procédural peut

avoir besoin des patrons Héritage, Encapsulation, Polymorphisme. L'existence des langages à objets, émulant ces patrons par leurs mécanismes, rend leur usage occasionnel.

Ce terme de relativité est à distinguer de la notion de subjectivité des patrons de conception. Il arrive en effet qu'un programmeur, par sa formation et son expérience personnelle, ne considère pas certains usages comme des patrons mais comme des bonnes pratiques inhérentes aux langages ou au paradigme. Ce programmeur a en quelque sorte « intégré » le motif dans son processus de développement courant.

2.2.1.2 Organisation initiale du catalogue *GoF*

À défaut de disposer d'une méthode d'organisation totale du catalogue (tel un langage de patrons), le *GoF* propose une classification suivant deux critères : la finalité et la portée.

La finalité reflète le but du patron : création, structure ou comportement. C'est donc un moyen de trier rapidement entre les problématiques abordées par les patrons.

La portée reflète le mécanisme central mis en jeu par le patron : classe ou objet. Un patron de *classe* implique des relations entre classes, de nature statique : le motif utilise exclusivement l'héritage. Un patron *objet* fonctionne d'abord grâce à des relations entre objets, donc dynamiques : le motif utilise la composition d'objets (ce qui n'interdit pas l'héritage).

Seul le critère de portée donne une indication sur l'implémentation : héritage statique ou composition dynamique. Ce critère entraîne une asymétrie prononcée entre les deux catégories. Quatres patrons sur vingt-trois sont dans la catégorie *classe*.

D'autres organisations du catalogue sont utilisées mais non formalisées : aspects de variabilité, relations entre patrons. [Zimmer, 1995] a analysé ces relations informelles pour décrire trois catégories : utilisation, similarité et combinaison. Les catégories *utilisation* et *combinaison* se rapportent directement aux motifs : un motif peut utiliser un autre motif dans sa description ; deux motifs peuvent être combinés ensemble pour cumuler leurs effets.

Observant que la relation d'utilisation est la plus fréquente (quatorze relations dans le catalogue), Zimmer propose une nouvelle organisation en trois couches respectant la transitivité des relations d'utilisation. La distinction illustre une hiérarchisation des patrons les plus basiques et communs vers les patrons spécifiques à un domaine et réutilisant d'autres patrons (tableau 2.1).

2.2.1.3 Classification par proximité aux langages de programmation

[Gil et Lorenz, 1997] soutiennent l'idée selon laquelle les motifs sont des mécanismes des langages en devenir. Leur classification repose sur une échelle de proximité au langage à trois niveaux : le cliché, l'idiome et le cadet (tableau 2.2).

Un cliché est un usage direct et commun des mécanismes prévalents des langages. Dans le cas des langages à objet, il s'agit des patrons Command (objectisation d'un concept pour le manipuler comme entité de première classe), Template Method (spécialisation par redéfinition d'une méthode), Facade (encapsulation sous une interface dédiée).

Un idiome est un motif subsumé par des mécanismes connus mais encore peu répandus. Il s'agit des patrons Visitor (émulé par les multi-méthodes dans CLOS), Abstract Factory et Factory Method (séparation entre type abstrait et classe d'implémentation), Singleton, Prototype, Memento (persistence).

Un cadet est un motif non encore capturé dans un langage. Gil et Lorenz opère une distinction entre deux sous-catégories suivant le nombre d'objets impliqués dans le motif. Les motifs

basés sur un découplage entre deux objets sont des *relators*, illustrant des patrons de « pseudo-héritage ». Les motifs décrivant un graphe d'objets en relation sont des architectes, illustrant des patrons de « pseudo-composition ».

2.2.1.4 Discrimination par rapport aux langages de programmation

[Agerbo et Cornils, 1998] proposent aussi de trier les motifs par rapport aux langages de programmation. Le principe est cependant différent : il s'agit de discriminer entre *motifs fondamentaux* et d'autres catégories de motifs. L'objectif est de préserver les bénéfices des patrons en restreignant le nombre effectif servant au partage et à la diffusion.

Agerbo et Cornils définissent trois critères : la capture par un mécanisme langage, la similarité à un motif plus basique, l'application d'un principe objet standard. Les patrons passant au travers du crible de ces trois critères sont déclarés comme fondamentaux. Les autres patrons sont recueillis dans deux catégories respectives : patrons dépendants du langage et patrons apparentés à un patron fondamental. Le troisième critère exclue certains patrons en tant que redites des principes objets (tableau 2.3).

La comparaison effectuée par les auteurs avec les travaux précédents souligne que les deux classifications ne se recoupent pas, malgré la proximité des critères (la capture et le principe objet étant jugés équivalents aux critères appliqués par [Gil et Lorenz, 1997]).

2.2.1.5 Discussion sur la disparité

La relativité des motifs de conception indique que des langages aux abstractions hautes capturent plus de motifs. Ceci n'enlève pas, à notre avis, l'intérêt des patrons : pour un langage donné, nous pensons qu'il existe toujours un ensemble de patrons afférents ; de plus un patron véhicule et illustre les compromis liés à l'implémentation de telles solutions.

Ceci renforce l'intérêt pour une estimation de la difficulté d'implémentation des motifs. Un motif proche du langage est facile à implémenter tandis qu'un motif plus abstrait demande un travail supplémentaire.

Les classifications ci-dessus établissent une certaine hiérarchie des motifs concrets et simples vers les motifs abstraits et complexes (tableaux 2.1, 2.2 et 2.3), bien que leurs objectifs soient différents (faciliter la compréhension, comparer au langage, sélectionner).

Mais ces classifications ne se recoupent pas et montrent quelques divergences. Le motif Strategy est décrit comme typique, un cadet ou n'est pas considéré comme un motif. Le motif Adapter est décrit comme basique, un cadet ou encore non considéré comme un motif. Le motif Abstract Factory est décrit comme typique, un idiome ou encore un motif (patron) fondamental. Le motif Command est typique dans un cas et un cliché dans l'autre. Le motif Visitor est typique mais aussi dépendant du langage.

Si ces répartitions ne sont pas forcément contradictoires, elles montrent qu'il est difficile d'obtenir une vision unique des motifs et de leur relation aux langages de programmation. En dehors du critère sommaire de portée, le *GoF* n'établit pas de classification des motifs par rapport aux mécanismes. Gil et Lorenz remarquent en conclusion qu'il est difficile d'établir une telle classification en raison des définitions plus ou moins précises des motifs.

Du point de vue du programmeur, cet état de fait est bien entendu lié à l'usage prédominant du langage naturel dans les descriptions des patrons. Cet usage permet une grande souplesse dans la somme des détails décrits et laisse la liberté d'interprétation et d'adaptation au programmeur. Cependant la complexité de l'implémentation est cachée au premier abord.

Du point de vue du langage, les patrons permettent donc par principe de décrire n'importe quelle solution. Il en résulte cette disparité entre les niveaux d'abstraction des motifs. Toute analyse de l'ensemble des patrons – et des motifs en particulier – doit prendre en compte cette disparité pour l'évaluation des moyens qu'elle compte utiliser.

Tableau 2.1 – Hiérarchisation des motifs suivant Zimmer

Spécifique	Interpreter
Typique	Abstract Factory, Bridge, Builder, Chain of Responsibility, Command, Iterator, Observer, Prototype, State, Strategy, Visitor
Basique	Adapter, Composite, Decorator, Facade, Factory Method, Flyweight, Mediator, Memento, Proxy, Singleton, Template Method

Tableau 2.2 – Classification des motifs suivant Gil et Lorenz. Les auteurs classent Memento et Visitor en tant que cadets et idiomes mais ne font pas apparaître Flyweight et Iterator.

Cadet	Adapter, Bridge, Builder, Chain of Responsibility, Composite, Decorator, Interpreter, Mediator, (Memento), Observer, Proxy, State, Strategy, (Visitor)
Idiome	Abstract Factory, Factory Method, Memento, Prototype, Singleton, Visitor
Cliché	Command, Facade, Template Method

Tableau 2.3 – Classification des motifs suivant Agerbo et Cornils

Fondamental	Abstract Factory, Bridge, Builder, Chain of Responsibility, Composite, Decorator, Flyweight, Iterator, Mediator, Memento, Proxy, State
Apparenté	Interpreter (à Composite), Observer (à Mediator)
Langage	Command, Facade, Factory Method, Prototype, Singleton, Template Method, Visitor
Exclus	Adapter (réutilisation), Strategy (liaison dynamique)

2.2.2 Problèmes dans l'implémentation des motifs de conception

Malgré la disparité des motifs de conception, des problèmes d'implémentation communs à la plupart des motifs de conception existent. Ceux-ci ont été principalement décrits dans [Soukup, 1995] et [Bosch, 1998].

2.2.2.1 Absence de traçabilité

Soukup remarque que, en raison de leur nature informelle, les patrons sont uniquement utilisés durant la phase de conception. Lors de la phase d'implémentation, les motifs sélectionnés sont perdus : ils deviennent invisibles.

Bosch utilise le terme *traçabilité* pour identifier ce problème : celui-ci insiste sur (la perte du) le lien entre le niveau conceptuel et l'implémentation. Faute de représentation du concept de motif dans les langages de programmation, il note que leurs implémentations sont dispersées dans plusieurs méthodes et classes.

Les conséquences concernent la maintenance de l'application. Soukup remarque que la compréhension est gênée car l'implémentation ne reflète pas la conception par les motifs. L'ajout et le retrait d'un motif est une tâche ardue car ses éléments dispersés sont difficiles à identifier. Le seul moyen de retrouver leurs traces est d'assurer un suivi de la documentation et des commentaires dans le code. Mais même cette documentation peut devenir obsolète si les programmeurs perdent de vue l'implication des motifs.

2.2.2.2 Interdépendances de classes

Soukup note que le comportement d'un motif implique des appels entre les classes impliquées dans le motif. Il se crée donc des dépendances entre les classes. Quand plusieurs motifs sont appliqués sur le même groupe de classe, il y a un risque d'interdépendances : chaque classe dépend, par transitivité, de toutes les autres. Ceci est une violation du *principe des dépendances acycliques* [Martin, 2000] à l'échelle des classes et nuit à la modularité de l'application.

Les conséquences concernent les phases de test et de maintenance. Il devient impossible de tester unitairement les classes du système, rendant plus difficile le débogage. La compréhension du fonctionnement d'une classe ne peut se faire sans parcourir de nombreuses autres classes.

Discussion : dépendance invasive des motifs ? Nous notons que l'ajout d'un motif dans une conception existante crée des dépendances supplémentaires, lesquelles ne vont pas forcément dans le sens des dépendances de la décomposition initiale. Ces dépendances invasives, liées à l'implémentation du motif, peuvent entraîner des interdépendances entre les rôles du motif.

Un exemple typique est le patron Observer. Celui-ci explique la dépendance de multiples observateurs vis-à-vis d'événements affectant un sujet : la dépendance « fonctionnelle » est donc des observateurs vers le sujet. Mais le motif implique l'appel par le sujet d'une méthode dans l'interface de l'observateur : la dépendance implémentée est donc inversée. Les observateurs ayant par ailleurs accès au sujet pour récupérer des données, une interdépendance entre sujet et observateur est créée. Le *GoF* indique qu'un tel couplage doit rester abstrait, par exemple grâce à une dépendance *via* une interface Java.

2.2.2.3 Réutilisation

Un patron permet déjà de « réutiliser » l'expérience en la capturant et la partageant. Le problème abordé ici est la réutilisation des motifs de conception et de leur implémentation. Ce problème concerne le passage de la version abstraite, décrite dans le patron, à une implémentation concrète, ainsi que les étapes intermédiaires potentielles.

Réutilisation des motifs abstraits. Suivant Bosch, la dispersion des implémentations des motifs à travers plusieurs classes empêche cette réutilisation. Par défaut de représentation du motif dans le code, il n'est pas possible d'en exprimer une forme réutilisable.

Soukup est plus précis en posant la distinction entre *abstract pattern* et *concrete pattern*. Un *abstract pattern* est la solution décrite dans le patron, indépendante du langage. Cette notion correspond à notre définition du motif de conception. Un *concrete pattern* est l'implémentation d'un motif dans un langage donné. L'objectif de Soukup est la définition d'une bibliothèque de motifs concrets réutilisables.

Nous posons donc le problème majeur de la réutilisation comme celui du passage du catalogue des motifs à une bibliothèque de solutions réutilisables dans un langage donné. Il s'agit en particulier d'obtenir des solutions aussi génériques que possibles.

Variations internes au motif. Un problème résultant de l'objectif de généricité est la prise en compte des variations internes à un motif. En effet chaque motif du *GoF* expose souvent plusieurs choix d'implémentations, à adapter suivant le contexte. Ces choix peuvent jouer sur la généricité et la dispersion de l'implémentation : il s'agit par exemple de décider de l'implémentation d'une interface par une classe et de la spécificité de cette interface.

Réutilisation d'implémentations concrètes. Un second objectif est la réutilisation d'implémentations concrètes, c'est-à-dire de solutions concrètes et déjà spécialisées pour une application particulière. C'est le problème classique abordé par les mécanismes d'héritage et de mixin. Cependant un problème résultant et plus particulier est le besoin d'un polymorphisme de famille. Étant donné la dispersion des motifs sur plusieurs classes, il faut assurer que les éléments du motif sont réutilisés en même temps.

2.2.2.4 Surcoût d'implémentation

Bosch remarque que de nombreux motifs nécessitent la répétition, moyennant adaptation, d'implémentations triviales, telles que la délégation d'un message. En Java cela peut aussi être, par exemple, l'itération par une boucle `for` sur une structure composite. Ce problème dépend fortement des langages et des facilités que ceux-ci offrent. En leur absence ou en l'absence d'outils de génération, la tâche est fastidieuse et la présence de ces artefacts répétés rend la maintenance plus difficile.

2.2.2.5 Problème d'identité

Le problème d'identité (*self problem*) est lié à l'utilisation de la délégation dans les classes par opposition à la délégation des langages à prototype [Lieberman, 1986]. Bosch note que ce problème se pose potentiellement à de nombreux motifs qui font usage de la délégation entre une classe interface (client) et une classe d'implémentation : l'identité du client est perdue par l'objet d'implémentation. Toute référence à `this` (ou `self`) fait référence à l'implémentation et non au client destinataire du message original.

Extension du concept. De plus, il existe le risque de révéler l'implémentation par passage de la référence de cet objet (en valeur de retour par exemple).

Un cas particulier de ce problème est la substitution d'identité. Le patron DECORATOR s'intéresse aux modifications dynamiques du comportement d'un objet décoré, sans que cela ait un impact sur le type de l'objet et des références à cet objet. Or le motif décrit l'encapsulation de l'objet décoré par un objet décorateur, ce qui implique de substituer toute référence au premier par une référence au second. Toute application se basant sur la référence pour établir l'identité d'un objet serait compromise par cette substitution.

Plus généralement le problème est de décider quel objet détient le `this` (ou `self`) lors de l'exécution. Le programmeur doit actuellement juger au cas par cas l'utilisation des références de ses objets, ce qui complexifie les phases de conception et de maintenance.

2.2.2.6 Bilan des problèmes d'implémentation

Ces problèmes communs à la plupart des motifs de conception illustrent, dans des variantes plus spécifiques, quelques-uns des problèmes récurrents des langages de programmation. Nous pouvons isoler deux surensembles distincts parmi ces problèmes : la modularité et l'expressivité.

D'une part, le manque de modularité marque les problèmes de traçabilité, d'interdépendance de classes et de réutilisation. L'absence de représentation de première classe des motifs est un obstacle à leur visualisation et manipulation dans le code. L'impact concerne toutes les phases en relation avec l'implémentation, de la conception à la maintenance.

D'autre part, le surcoût d'implémentation et le problème d'identité, décrits par Bosch, sont fortement dépendants de l'expressivité du langage choisi. Le problème d'identité souligne par exemple un mécanisme purement objet, lié à la délégation.

Si les trois premiers problèmes peuvent être abordés dans une réflexion commune sur la modularité, nous nous intéresserons de façon accessoire aux deux autres problèmes.

2.2.3 Composition des motifs de conception

La composition des motifs de conception est un sujet peu étudié au sein du domaine des patrons de conception. Cette composition se traduit actuellement par une juxtaposition souvent ad hoc des implémentations des motifs.

Nous examinons plusieurs travaux exposant différentes notions de composition. Pour illustrer les enjeux potentiels de la composition, nous commençons par présenter la notion de densité des motifs.

2.2.3.1 Notion de densité des motifs

Le *GoF* se termine par une citation du livre de Chistropher Alexander, *« A Pattern Language »* :

> *"It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this, is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound."*

Il est délicat d'interpréter une telle citation hors de son domaine, encore plus de la mettre en relation avec un domaine a priori étrangé comme l'informatique. Nous retenons qu'il existe une

notion de densité[4] et qu'une forte densité indique un mélange des motifs et produit du « sens ». Il est facile d'imaginer la notion de densité des motifs transposée aux programmes, beaucoup moins de voir quel sens s'en dégage. Un programmeur n'est pas obligé de comprendre ni même de connaître les motifs impliqués : ceux-ci ne définissent pas la sémantique du programme, mais peuvent en donner une interprétation plus abstraite. Le *GoF* reste évasif sur le sujet : il s'agit d'abord d'une réflexion et d'une invitation à approfondir le domaine, prédisant que la somme de motifs devraient donner de meilleures conceptions.

[Gamma et Beck, 2002] donnent une illustration plus précise de la notion de densité dans le *« JUnit Cook's Tour »*. Cet article présente la conception du cadriciel JUnit de façon incrémentale et en utilisant les patrons de conception pour résoudre les difficultés. Le résultat final présente un noyau de cinq classes dans lequel six motifs sont impliqués. Dans les conclusions de cette étude empirique, Gamma et Beck soulignent par ailleurs la forte densité des motifs autour de la classe `TestCase`, abstraction clé du cadriciel. Cette densité est pour eux un signe de maturité du cadriciel (la maturité étant vue comme une qualité, au même titre que le sens dans le propos d'Alexander). Ils remarquent qu'une forte densité rend la conception plus facile à utiliser mais plus difficile à changer. Malheureusement cette affirmation abordée dans la conclusion n'est pas développée par l'article.

2.2.3.2 Relations entre motifs de conception

Le *GoF* indique différentes relations entre les patrons et les motifs. Le format de description inclut une rubrique *« Related Patterns »* contenant, pour chaque patron, de brèves mentions sur les relations avec d'autres patrons. Le *GoF* affiche aussi un graphe des relations [Gamma *et al.*, chapitre 1, page 12]. Ces relations restent décrites de façon succincte et informelle.

[Zimmer, 1995] propose une classification de ces relations en trois catégories :

- utilisation : d'un motif par un autre (où le premier sert à résoudre un problème dans l'implémentation du second) ;
- similarité : des problèmes des patrons (dans le but de comparer les motifs et les compromis proposés par les patrons concernés) ;
- combinaison : de motifs (collaboration de motifs – contrairement à l'utilisation, les deux motifs ne sont pas confondus dans leurs implémentations respectives).

Analyse. Les catégories *utilisation* et *combinaison* se rapportent toutes les deux à la composition des motifs. Le premier usage est confondu avec l'implémentation standard des motifs. La combinaison est plus intéressante en terme de composition modulaire. Les combinaisons citées sont :

- Factory Method et Template Method ;
- autour de Composite : Iterator et Visitor, Decorator, Builder ;
- Chain of Responsibility et State.

Ces relations ne donnent pas d'information sur la façon dont ces motifs se composent. C'est cependant une première étape qui permet d'identifier les collaborations les plus courantes.

[4] Intuitivement, la densité représente le rapport entre une « quantité » de motifs et une « quantité » de classes recouvrant ces motifs.

2.2.3.3 Étude de composition de motifs

L'étude approfondie d'une composition de motifs est l'étape suivante.

[Braux et Noyé, 1999] proposent une composition des motifs STRATEGY et OBSERVER donnant une architecture appelée *ComportementOuvert*. Leur but est de gérer les changements dynamiques de comportement d'un ensemble d'objets interdépendants. Chaque objet est un observateur des états d'un sous-ensemble d'objets : un changement d'état dans ce sous-ensemble détermine un changement d'état de l'objet local. Ce dernier adapte alors son comportement grâce à une implémentation interne du motif STRATEGY.

Braux et Noyé indiquent que le résultat de cette composition est en fait une variation autour du patron STATE, lequel aborde le problème du changement dynamique de comportement d'un objet en fonction de son état interne. Le niveau d'abstraction interne au motif STATE est cependant variable, ce qui complique son implémentation : la gestion des transitions est peu détaillée car plusieurs options existent [Gamma *et al.*, page 305, rubrique Implémentation]. La composition de STRATEGY et OBSERVER raffinent l'implémentation d'un motif STATE en détaillant ce problème.

Analyse. Les motifs STRATEGY et OBSERVER proposés par Braux et Noyé sont des versions spécialisées des motifs originaux. Leur composition est donc l'implémentation *ad hoc* d'une collaboration. Braux et Noyé note que l'exploitation de cette collaboration permet une approche systématique du problème visé, ce qui ouvre la voie à la génération de code.

2.2.3.4 Techniques structurelles et comportementales pour la composition

[Yacoub et Ammar, 2003] étudie plusieurs techniques pour la composition des motifs de conception. Un point intéressant dans cette étude est la distinction entre deux catégories de techniques : les techniques comportementales et les techniques structurelles.

Les techniques comportementales, selon Yacoub et Ammar, regroupent les travaux se focalisant sur l'abstraction du comportement des objets impliqués dans les motifs. La modélisation des rôles des objets (participants d'un motif) est un prémice intéressant et a été exploité pour la composition par [Riehle, 1997] et [Bosch, 1998]. D'une manière générale, ils remarquent que la formalisation des motifs, par exemple par des *contrats*, est une étape préliminaire intéressante dans l'étude de la composition. Les travaux sur la composition proprement dite manquent.

Les techniques structurelles regroupent les travaux se focalisant sur la modélisation et la réutilisation des structures des motifs, c'est-à-dire principalement des diagrammes de classe. Les travaux cités concernent notamment la composition au niveau conception utilisant UML.

Analyse. Le focus de Yacoub et Ammar est sur la conception d'applications par composition de motifs. Les motifs sont vus comme des briques de base du programme. Cette approche est volontairement indépendante du code, utilisant la notation UML. Les problèmes liés à l'implémentation des motifs et de leurs compositions sont donc ignorés.

2.2.3.5 Composition au niveau langage

[Soukup, 1995] relève le problème d'interdépendances des classes (section 2.2.2.2) lié à l'implémentation de plusieurs motifs dans un même ensemble de classes. Ce problème fait écho à la composition des motifs et à la difficulté d'exprimer cette composition dans les langages actuels.

Le modèle *LayOM* [Bosch, 1998] est basé sur l'encapsulation d'un objet dans une série de couches (*layer*). Chaque message reçu (resp. envoyé) est intercepté par la couche extérieure (resp. intérieure), traité et passé à la couche immédiatement inférieure (resp. supérieure) jusqu'à l'objet. Ce mécanisme est appelé superimposition. L'ordre statique de déclaration des couches définit l'ordre de composition.

[Bosch, 1998] montre plusieurs extensions de langage permettant de faciliter l'implémentation dans des couches des rôles de différents motifs. En conséquence, ce modèle permet la composition de plusieurs rôles de différents motifs dans le même objet.

2.2.3.6 Composition au niveau conception

[Riehle, 1997] présente la notion de *Composite Design Patterns*. Ces motifs composites capturent la synergie de plusieurs motifs collaborant ensemble. Riehle propose une méthodologie de conception pour la capture de ces motifs composites.

Un motif est décomposé en un ensemble de rôles (participants) et de contraintes entre ces rôles. Ces contraintes permettent de définir une matrice indiquant, pour chaque rôle, si l'objet correspondant doit, peut ou ne doit pas implémenter un autre rôle dans le motif. La composition de motifs consiste alors à composer leurs matrices et propager les contraintes entre rôles. Les rôles aux contraintes équivalentes sont ensuite fusionnés pour simplifier la matrice. Cette matrice illustre les rôles et les contraintes du nouveau motif composite.

Le motif composite est plus simple car il utilise un vocabulaire réduit tout en gardant la synergie de la collaboration. Ce travail se situe cependant uniquement au niveau conceptuel : il ne s'agit pas d'explorer les problématiques liées à l'implémentation.

2.2.3.7 Bilan sur la composition

Il n'y a pas de définition unique sur ce qu'est concrètement la composition des motifs de conception. La notion la plus universelle est que la composition est une collaboration (sections 2.2.3.2, 2.2.3.3) ou une synergie (section 2.2.3.6) de plusieurs motifs. Ces travaux suggèrent que les motifs ont besoin d'être spécialisés ou adaptés pour une composition particulière.

Les travaux sur la composition se situent à des niveaux d'abstraction différents : certains s'intéressent uniquement au niveau conception de la composition en se basant sur des diagrammes de rôles ou de classes. La composition via un langage a été abordée (section 2.2.3.5) mais peu étudiée. Par conséquent l'implémentation de la composition des motifs se résume souvent à une juxtaposition ad hoc, spécialisée dans le code existant (section 2.2.3.3).

Pourtant la notion de densité des motifs de conception (section 2.2.3.1) illustre l'occurrence de ces compositions dans des applications et des cadriciels aboutis. Les conséquences, tant positives que négatives, restent mal connues.

Or l'étude de la composition des motifs de conception est un préliminaire à la compréhension des effets de la densité. Les travaux illustrent une progression depuis les premières classifications des relations aux tentatives générales des langages et des méthodes de conception, en passant par les études de cas.

Mais nous pensons que cette difficulté dans la définition et l'étude de la composition des motifs dépendent fortement des points précédents :

- l'absence de modularité des implémentations (section 2.2.2.6) gêne l'analyse de l'impact d'une composition ;

- le caractère informel et la disparité des motifs (section 2.2.1.5) rendent difficile toute généralisation.

2.3 Approches pour l'implémentation des motifs de conception

[Chambers *et al.*, 2000] discute les relations entre patrons de conception, langages de programmation et outils. L'article débat de l'apport des patrons et du support potentiel apporté par les langages de programmation et les outils. Bien que l'article confronte les approches, deux points font consensus : l'utilisation des patrons pour communiquer et discuter les problèmes de conception ; le besoin de traçabilité des motifs dans le code.

Ce dernier point, discuté en section 2.2.2.1, est traité différemment par les langages et les outils. L'approche des langages consiste à créer ou modifier des mécanismes pour subsumer les motifs, ou au moins faciliter leur implémentation. L'approche des outils vise à compléter les langages de programmation par des représentations plus abstraites et des usages comme la génération ou la détection.

Dans la suite de ce chapitre nous présentons les travaux les plus significatifs de ces deux approches. Notre angle de vue est le traitement des problèmes d'implémentation des motifs vus en section 2.2.2.

2.3.1 Langages de programmation

Comme noté en section 2.2.1.1, la difficulté d'implémentation d'un motif dépend des mécanismes du langage utilisé. Le *GoF* fait explicitement usage des mécanismes les plus standards des langages à objets. Les travaux présentés ci-dessous ciblent spécifiquement les motifs de conception.

2.3.1.1 Langages et bibliothèques de motifs

[Chambers *et al.*, 2000] répertorie plusieurs mécanismes améliorant les implémentations et expérimentés dans Dylan et Cecill. [Norvig, 1998] fait la démonstration de ces mécanismes dans des langages dynamiques comme Lisp et Dylan. [Agerbo et Cornils, 1998] montrent comment le langage Beta change l'implémentation de plusieurs motifs courants. Agerbo et Cornils proposent, grâce à ces changements, de constituer des bibliothèques de motifs réutilisables, faciles à tracer.

Cependant ces mécanismes n'ont toujours pas fait leur chemin dans les langages les plus courants. Les problèmes classiques de conception des langages se posent : interactions avec les autres mécanismes, concessions liées au langage, gestion de la compabilité avec l'existant. Le problème est aussi de choisir quels motifs sont à intégrer et lesquels sont laissés dehors car trop simples ou trop compliqués (section 2.2.1.3).

[Arnout, 2004] fait la démonstration dans le langage Eiffel du processus de *componentization*, c'est-à-dire de la réécriture des motifs en tant que composants dans des bibliothèques prêtes à l'emploi. Les mécanismes propres à Eiffel (par exemple, la généricité, les tuples et les agents) sont mis à contribution pour écrire pour chaque motif une bibliothèque dont le comportement émule ce motif. Les résultats obtenus sont encourageants puisque 65% des motifs du *GoF* sont capturés, au moins partiellement, dans des bibliothèques.

Cependant ce travail se heurte aussi à la disparité des motifs : toutes les variations d'un motif ne sont pas nécessairement disponibles dans la bibliothèque associée ; certains motifs sont trop

dépendants du contexte de l'application pour être définis dans une bibliothèque – un générateur de squelette du motif est alors proposé. Il serait intéressant d'établir quels mécanismes du langage Eiffel sont essentiels à ce processus et pourraient être transposés à d'autres langages comme Java ou C++.

2.3.1.2 Extensions de langage

FLO est un modèle explicitant les interactions entre objets [Ducasse, 1997]. Il définit deux concepts : les connecteurs et les règles d'interactions. Les connecteurs sont des objets représentant des relations entre des participants, rôles joués par des instances de classes. Les connecteurs déclarent un ensemble de règles d'interactions. Ces règles interceptent les messages reçus par les participants : elles réagissent par une action (inhibition, transmission, nouveau message envoyé à un des participants) si le message est filtré par la règle.

Le modèle FLO permet d'implémenter le comportement et la structure de certains motifs dans un connecteur. Celui-ci devient donc une représentation du motif, éliminant le problème de traçabilité. Les dépendances entre classes des participants sont aussi réduites puisque capturées dans le connecteur. Les règles d'interactions éliminent le surcoût d'implémentation lié à la propagation de messages. Le modèle est extensible grâce à un protocole de méta-objet. Cependant les trois règles de base sont proposées comme suffisantes pour la plupart des cas, y compris pour l'implémentation des motifs.

Bien que cette approche soit prometteuse en raison de sa polyvalence et de sa simplicité, elle n'a pas été étudiée dans des applications à forte densité de motifs ni reprise dans d'autres langages.

Le modèle *LayOM* (*Layered Object Method*) est un modèle objet étendu avec les concepts de couches encapsulant un objet et d'interception des messages par ces couches [Bosch, 1998]. *LayOM* est aussi un modèle extensible grâce à une architecture de compilateurs délégués : il permet de définir de nouveaux types de couches et traitements des interceptions.

Bosch étend son modèle avec de nouvelles couches pour implémenter le comportement des motifs dans les objets. Par exemple le motif Observer est implémenté par une couche `Observer` à appliquer sur les sujets. Le motif est utilisé et paramétré grâce à une syntaxe spécifique à la couche `Observer` (source 2.1). La couche définit la sémantique et l'infrastructure associée (gestion des observateurs).

Exemple 2.1 – Usage du motif Observer dans le modèle *LayOM*

```
1  class ObservablePoint
2   layers
3     st : Observer(notify after on setX, notify after on setY);
4   methods
5     setX(newX : Location) returns Location
6        begin ... end;
```

Cette approche permet de localiser dans chaque classe ou objet la définition du motif. L'aspect déclaratif et la délégation de compilateurs évitent le surcoût d'implémentation ainsi que la multiplication des classes dédiées aux motifs. Cependant le modèle *LayOM* permet la délégation

ou la redirection des messages mais pas le traitement des paramètres. Cette approche tend aussi à encombrer la syntaxe : chaque motif définit sa propre syntaxe tout en masquant la sémantique.

2.3.1.3 Programmation générative

[Tatsubori et Chiba, 1998] montre l'utilisation d'un protocole de méta-objet (MOP) pour adapter les classes C++ aux motifs lors de la compilation. Ce MOP est extensible : il permet de définir des méta-classes s'exécutant à la compilation pour modifier les classes ciblées. Il permet aussi de définir une syntaxe spécifique à chaque méta-classe. Cette syntaxe permet la déclaration par les classes ciblées des opérations et paramètres de modifications.

Le principe de conception du MOP implique de programmer l'implémentation d'un motif générique dans une méta-classe. Cette approche est destinée aux problèmes de traçabilité, de surcoût d'implémentation et de réutilisation. Elle n'a cependant pas été évaluée sur un programme plus large.

2.3.2 Outils de programmation

Les outils intègrent les patrons au cycle de programmation. Un outil utilise une bibliothèque de modèles des patrons (ou des motifs) plus ou moins proches du langage cible. Le modèle utilisé dépend aussi de l'usage visé par l'outil : transformation de code ou détection.

2.3.2.1 Transformation de code

La transformation de code, dans son sens le plus large, consiste à produire automatiquement l'implémentation d'un motif à partir de spécifications. La forme prise par ces spécifications dépend de l'outil et de son fonctionnement. Il peut s'agir de spécifications abstraites issues de la conception : dans ce cas l'outil génère un code à partir d'un modèle abstrait du motif paramétré avec ces spécifications. Mais il peut s'agir aussi de spécifications extraites d'un code existant : dans ce cas l'outil réécrit le code dans une démarche de restructuration, en veillant à respecter les spécifications existantes.

Les outils de transformation de code ont le potentiel pour résoudre les problèmes de réutilisation et de surcoût des implémentations des motifs. Cependant ce domaine de recherche est aussi confronté à d'autres difficultés : la disparité et le côté informel des motifs est un obstacle à la définition de modèles précis ; un même motif peut présenter de nombreuses variations, rendant encore plus complexe sa modélisation ; enfin ces outils ne traitent pas forcément la traçabilité des modifications manuelles du code. Ce dernier point expose un défaut commun à de nombreux outils de génération : l'écrasement de ces modifications en cas de regénération. Les outils de réécriture, par principe, sont moins exposés à ce problème.

Nous donnons un aperçu de ces deux types d'outils avec les deux exemples ci-dessous.

Classes de motif. [Soukup, 1995] définit dans une bibliothèque des « classes de motif » réutilisables. Le principe est de ne pas polluer les classes participantes avec les méthodes du motif. Pour cela une classe de motif modularise l'interface publique et le comportement interne du motif : la solution objet originale du motif est donc transformée. Les classes participantes ne font que définir les données propres à leur utilisation dans le cadre du motif (variables d'état et de relation avec d'autres instances).

Pour utiliser cette bibliothèque, le programmeur doit déclarer les motifs à implémenter grâce à des directives paramétrées avec le nom des classes participantes. Les classes participantes doivent aussi déclarer des directives, spécifiques à chaque motif, afin d'implémenter les données propres au motif.

Un préprocesseur lit les directives du programme et des classes participantes afin de générer le code correspondant : celui-ci consiste en la définition des classes de motifs spécialisées pour l'occurrence et de macros (en C++) modifiant les classes participantes.

La définition de la bibliothèque proprement dite est une affaire de métaprogrammation : il s'agit de définir des motifs génériques, paramétrés à la fois par les noms des participants et par des macros permettant la spécialisation du motif.

L'intérêt de cette approche, outre la réutilisation, est la localisation d'une grande partie de l'implémentation du motif dans la classe de motif. Cette approche améliore la traçabilité en isolant chaque occurrence de motif et réduit les dépendances entre classes participantes. Elle suppose cependant que le motif générique fourni par la bibliothèque est en adéquation avec les contraintes du programme : il n'est en effet pas possible de modifier l'implémentation du motif en dehors de la bibliothèque.

La réécriture de code est basée sur des règles automatiques de réécriture d'expressions élémentaires du code. Ces règles transforment une caractéristique particulière de l'expression tout en préservant ses autres propriétés (contrairement à la génération). Ces règles sont à la base des méthodes (formalisées ou non) de restructuration [Fowler *et al.*, 1999]. Elles sont de plus en plus intégrées à des outils comme les environnements de développement intégré (EDI).

Cependant la recherche des règles de réécriture pour implémenter des motifs de conception est un domaine naissant. Le projet Lutin [Ziane, 2004] développe des règles de pliage visant à implémenter les mécanismes sous-jacents à la plupart des motifs de conception, à commencer par le découplage via l'indirection de méthodes. La combinaison de ces règles permet de reproduire différentes variantes pour plusieurs patrons. Un autre objectif de ce projet est l'identification et la vérification de contraintes de qualité que les patrons abordent dans leur description. Cette démarche supporte donc la traçabilité des motifs en capturant les contraintes et les choix menant à leur implémentation.

2.3.2.2 Détection de motifs

La détection de motifs représente en quelque sorte la démarche inverse de la génération de code. Il s'agit de partir du code existant afin d'y détecter des motifs perdus ou potentiels. L'objectif est l'identification et la localisation dans le code des motifs : ce résultat peut être exploité dans un but de documentation, d'évaluation de la qualité ou de la complexité du programme, ou encore comme guide pour la restructuration du programme en vue de l'améliorer. Cette démarche s'inscrit donc dans les phases d'implémentation et de maintenance du programme. Elle prend le problème de la traçabilité « à l'envers ».

[Albin-Amiot, 2003] présente le métamodèle PDL (*Pattern Description Language*) pour décrire les motifs de conception à des fins de génération de code et de détection des formes *complètes* des motifs. En pratique les motifs se retrouvent souvent dans le code sous une forme incomplète ou déformée. [Guéhéneuc, 2003] raffine PDL avec PADL (*Pattern and Abstract-level Description Language*) pour pouvoir décrire avec plus de détails les implémentations des motifs et des programmes. Ce modèle est en effet exploité pour analyser le code source d'un programme de

façon statique et dynamique. La détection d'un motif est définie comme un problème de satisfaction de contraintes sur le modèle du programme. Le solveur de contraintes exécute la détection proprement dite : en fonction de contraintes plus ou moins relâchées, les candidats les plus probables au rang de motifs sont identifiés. Ces données peuvent être ensuite exploitées dans la rétroconception, la documentation ou la restructuration du code.

2.4 Conclusion

Nous avons présenté dans ce chapitre les difficultés de l'implémentation des motifs de conception. D'abord la disparité des motifs (c'est-à-dire les niveaux d'abstraction et de complexité variables entre chaque motif) gêne toute généralisation à partir de modèles propres à automatiser la conception ou la programmation avec les motifs. La description informelle, en langage naturel, laisse une grande liberté à l'interprétation du programmeur. Ensuite les problèmes d'implémentation des motifs (traçabilité, réutilisation et interdépendances) sont liés à leur absence de modularité. Enfin cet absence de modularité gêne aussi l'étude de la composition des motifs et de ses conséquences sur le code, réduits à une juxtaposition ad hoc. Pourtant la notion de densité souligne que les motifs de conception et leurs compositions ont une importance croissante dans la maturation des applications.

Les langages et les outils de programmation sont deux domaines actifs dans la recherche et la résolution des problèmes liés à la modularité. Les langages peuvent fournir de nouveaux mécanismes subsumant les patrons – le motif devient alors obsolète et disparaît pour ce langage particulier. Cependant un langage peut aussi capturer les éléments du motif pour le modulariser et améliorer sa traçabilité. Les outils intègrent les patrons au cycle de programmation. Dans ce cadre certains outils visent la génération ou la réécriture de code pour pallier aux problèmes de réutilisation et de surcoût d'implémentation. D'autres outils visent la détection de motifs potentiels pour améliorer la traçabilité et faciliter la restructuration.

Dans cette thèse nous nous intéressons à l'approche des motifs de conception avec un nouveau paradigme de programmation. Nous retenons des travaux précédents les leçons suivantes :

- les mécanismes subsumant les motifs doivent être généraux et non spécifiques à chaque caractéristique d'un motif ;
- le modèle de programmation doit être simple et uniforme de façon à être assimilé rapidement ;
- l'adaptation permise par les motifs (malgré la complexité de leurs implémentations) doit être conservée après transformation.

Ces trois conditions sont à notre avis nécessaires pour l'adoption d'un nouveau mécanisme. Cependant même le modèle FLO [Ducasse, 1997] qui répond à ces trois conditions (section 2.3.1.2) n'a pas été exploité dans d'autres langages. Nous en concluons qu'un nouveau mécanisme doit avant tout chercher à résoudre un problème plus général que l'implémentation d'un motif de conception.

Chapitre 3

Approche des motifs avec la programmation par aspects

Sommaire

Ce chapitre présente l'approche de la programmation par aspects pour l'implémentation des motifs de conception. Un aspect est un module encapsulant une préoccupation dite transversale. Cette nature transversale désigne une préoccupation non modularisée dans une décomposition hiérarchique en classes. Son implémentation est affectée par la dispersion et le mélange du code avec d'autres préoccupations. La programmation par aspects fournit de nouveaux mécanismes pour modulariser ces préoccupations en complément des mécanismes traditionnels.

La dispersion des implémentations et les problèmes liés à la modularité des motifs en font des candidats intéressants au titre d'aspects. L'approche des aspects a deux caractéristiques complémentaires : certains éléments de l'implémentation du motif sont transformés grâce aux nouveaux mécanismes ; l'ensemble est modularisé dans un aspect.

3.1 Présentation de la programmation par aspects

La séparation des préoccupations est un principe fondamental du génie logiciel [Dijkstra, 1974]. Elle s'appuie généralement sur la modularité, une propriété recherchée dans le développement

des programmes. Les caractéristiques fondamentales d'un module sont la séparation entre interface et implémentation, l'encapsulation de l'implémentation et la composition de modules via leurs interfaces.

Les techniques de programmation actuelles (langages, outils, méthodes) proposent des mécanismes pour assurer la modularisation des préoccupations : procédures, interfaces, classes. Cependant la décomposition modulaire autorisée par ces mécanismes est hiérarchique : elles privilégient une organisation du programme en couches, où chaque nouvelle couche dépend de la précédente. Dans la suite de ce chapitre, nous utiliserons le qualificatif de hiérarchique pour désigner ces techniques.

Or toutes les préoccupations d'un programme ne sont pas forcément modularisées dans une décomposition hiérarchique. Ces préoccupations non modularisées sont dites transversales. Conceptuellement, un aspect est un module encapsulant une préoccupation transversale [Kiczales *et al.*, 1997].

Cette section détaille ces concepts à travers une grille de lecture des langages d'aspect et leurs incarnations dans AspectJ.

3.1.1 Concepts essentiels de la programmation par aspects

La programmation par aspects se définit, du point de vue du problème, comme une technique visant la modularisation des préoccupations transversales.

La solution proposée par la programmation par aspects est l'expression de modules liées à des interfaces transversales aux couches hiérarchiques. Cette technique vient donc en complément de la décomposition hiérarchique. Trois concepts sont essentiels pour décrire cette solution.

- Un point de jonction (*join point*) est un point intéressant du programme.
- Une coupe (*pointcut*) est une description abstraite d'un ensemble de points de jonction.
- Un aspect est un module associant un ensemble de coupes à un ensemble d'actions implémentant une préoccupation transversale.

Suivant le point de vue des aspects, un programme (resp. son exécution) est représenté(e) par un graphe (resp. une séquence) de points de jonction. Les coupes permettent de sélectionner dans ce graphe un sous-ensemble de points intéressants pour une préoccupation : à ces points sont exécutées les actions pertinentes implémentant la préoccupation. Les coupes d'un aspect définissent l'interface transversale, indépendantes de la décomposition en couches.

3.1.2 Problématiques liées aux préoccupations transversales

Les retours d'expérience des années 80, en particulier en programmation par objets, ont permis d'identifier la rigidité liée à la décomposition hiérarchique comme source de problèmes. Les travaux décrits ci-dessous ont participé à cette identification et à la formation de la communauté des aspects [Lopes, 2005].

[Ossher et Tarr, 2000] ont proposé le concept de séparation des préoccupations dans plusieurs dimensions. Dans cette vue conceptuelle, la décomposition hiérarchique est critiquée car elle aboutit à une solution unique pour décrire le programme et son fonctionnement. Or cette solution n'est pas forcément adaptée à l'expression de toutes les préoccupations du programme. Leur implémentation est handicapée par le modèle dominant de la conception. Ossher et Tarr parlent de tyrannie de la décomposition dominante car il est difficile, une fois cette décomposition faite, de changer de modèle de fonctionnement.

La programmation par aspects [Kiczales *et al.*, 1997] offre une vue plus technique de la décomposition hiérarchique et de ses effets. Quelque soit la décomposition d'un programme dans une hiérarchie de modules, le principe des aspects postule que certaines préoccupations, dites transversales, ne peuvent être modularisées complètement. Cette transversalité se manifeste dans l'implémentation à travers deux symptômes : la dispersion du code d'une préoccupation à travers plusieurs modules ; le mélange du code de plusieurs préoccupations dans un module.

La programmation adaptive [Lieberherr *et al.*, 2001] recommande une liaison plus souple entre la spécification d'un programme et son implémentation. Ce concept est basé sur la loi de Demeter : « ne communiquez qu'avec vos amis », ce qui signifie pour un objet de n'envoyer des messages qu'à lui-même, ses composants et ses paramètres. La violation de cette loi implique d'avoir accès à des objets hors du contexte de l'objet courant. Or cet accès, même s'il se fait via les interfaces, entraîne des dépendances à d'autres classes (donc d'autres modules) : le chaînage des accès révèle une partie de leur implémentation, violant l'encapsulation des modules. Lieberherr propose le concept de traversée, un descripteur de parcours à travers un réseau d'objets faisant abstraction des détails d'implémentation.

3.1.3 Mécanismes des langages à aspects

Les langages à aspects proposent des mécanismes capturant ces concepts essentiels. Ils sont conçus pour être utilisés en complément d'un autre langage, appelé langage de base. La liste des mécanismes détaillée ci-dessous constitue une grille de lecture pour l'analyse des caractéristiques d'un langage à aspects [Cointe *et al.*, 2005].

- Modèle des points de jonction
- Langage de coupe
- Langage d'action
- Aspect (module)
- Tissage

Nous utilisons AspectJ [Kiczales *et al.*, 2001] pour illustrer ces mécanismes. AspectJ est une extension du langage Java. Il permet de programmer des aspects en Java pour des programmes Java. Il a à ce titre le pouvoir d'expression d'un langage de programmation généraliste.

L'exemple 3.1 montre une occurrence basique du motif Observer réalisée avec AspectJ. La classe `Counter` peut incrémenter un compteur interne. À chaque modification de sa valeur, un événement est généré pour mettre à jour une vue, instance de la classe `CounterView` (non montrée). En Java standard ceci revient à insérer dans le code une notification à la vue (code commenté de la ligne 4). Avec AspectJ, les expressions du programme (appel de méthode, accès à un attribut) génère des points de jonction : l'événement de mise à jour est représenté par un point de jonction (exécution de la méthode `setValue` par exemple) que l'aspect peut capturer. Le but de l'aspect `UpdateView` est de modulariser la dépendance à `CounterView` dans `Counter` qu'induirait une notification explicite (ligne 4). Nous détaillons la mise en œuvre de cette solution avec AspectJ dans les rubriques suivantes.

3.1.3.1 Modèle des points de jonction

Le modèle de points de jonction est la vision abstraite des aspects vis-à-vis du langage de base. Il décrit les points dans l'exécution d'un programme susceptibles d'être interceptés par les aspects. Un programme se caractérise donc par la trace de points de jonction générée.

Exemple 3.1 – Exemple d'aspect dans le langage AspectJ

```
1  public class Counter extends Object{
2    private int value;
3    public void setValue(int nv){
4      value = nv;     // view.update(nv);
5    }
6    public void incr(int delta){
7      this.setValue(value+delta);
8    }
9
10   public static void main(String[] args) {
11     Counter c1 = new Counter();
12     c1.incr(1); c1.incr(6); c1.raz();
13   }
14 }
15
16 public aspect UpdateView {
17   private CounterView view;
18   pointcut changed(): call(void Counter.setValue(int));
19   after(int n, Counter r): changed() && args(n) && target(r){
20     view.update(r, n);   // after advice
21   }
22 }
```

AspectJ supporte une dizaine de points de jonction possibles : appel ou exécution de méthode ou de constructeur, accès à un champ en écriture ou en lecture, initialisation d'instance, interception d'une exception. Il faut noter un biais de ce modèle : AspectJ reconnaît uniquement les mécanismes objets de Java ; les structures de contrôle comme la conditionnelle et la boucle sont ignorées.

Enfin ce modèle n'est pas extensible en AspectJ. Il pose les limites de ce qu'il est possible d'intercepter directement.

3.1.3.2 Langage de coupe

Le langage de coupe permet la description abstraite d'un ensemble de points de jonction. Il est constitué de différentes opérateurs. Chaque type de point de jonction est capturé par une primitive, c'est-à-dire un opérateur atomique dédié. Les opérateurs de filtrage permettent de sélectionner un point de jonction suivant une caractéristique du programme sous-jacent au point (type de l'objet ou des arguments, nom de méthode...). Les opérateurs de composition permettent de combiner plusieurs opérateurs entre eux. Les opérateurs temporels ou relationnels filtrent les points de jonction en fonction de la structure de la trace (il peut s'agir par exemple de l'arbre des appels de méthodes).

Le rôle d'une coupe est aussi d'extraire des données du contexte correspondant au point de jonction. Il existe donc des opérateurs et des primitives d'accès au contexte. Ce contexte est ensuite exposé par la coupe.

Une coupe est donc soit une primitive, soit une composition d'opérateurs définie par le

programmeur.

AspectJ utilise, outre les primitives, trois opérateurs logiques pour la composition (`et` logique, `ou` logique et négation logique). Les opérateurs `this(), target(), args()` permettent le filtrage et l'extraction de contexte sur les arguments. L'opérateur `cflow()` est un opérateur relationnel sur le flot de contrôle : il prend en paramètre une autre coupe et capture tous les points de jonction se trouvant *dans* le flot de contrôle de cette coupe (typiquement l'appel ou l'exécution d'une méthode).

La ligne 18 (exemple 3.1) montre la coupe nommée `changed()` définie par une seule primitive : `call(void Counter.setValue(int))`. Cette primitive capture les appels de méthodes mais est paramétrée par un filtre qui cible uniquement les appels à la méthode `setValue(int)` de la classe `Counter`.

La ligne 19 montre la déclaration d'une coupe anonyme par la composition `changed() && args(n) && target(r)`. `args(n)` récupère le paramètre effectif d'appel dans la variable `n` ; `target(r)` fait de même avec l'objet cible de l'appel. La déclaration typée des paramètres formels `n` et `r` agit aussi comme un filtre : seuls les points de jonction où les paramètres effectifs ont un type compatible seront capturés.

3.1.3.3 Langage d'action

Le langage d'action caractérise la puissance d'expression disponible à chaque point de jonction capturé. Il peut être limité ou général. Une action est similaire à une méthode et caractérisée aussi par une signature : celle-ci comprend la coupe visée ainsi que les paramètres formels de l'extraction du contexte.

AspectJ distingue aussi le moment d'exécution de l'action par rapport au point de jonction : avant, après (un retour normal, un lancer d'exception ou toute exécution) ou à la place (`around`). Danc ce dernier cas, l'action remplace l'exécution du point de jonction et du flot de contrôle sous-jacent. Le programmeur peut cependant relancer ce point en faisant appel à l'opérateur spécial `proceed()`.

La déclaration d'une action (ligne 19) contient aussi l'association avec la coupe visée. `after(int n, Counter r)` indique une exécution après le point de jonction. Le corps de l'action (ligne 20) est écrit en Java standard.

3.1.3.4 Aspect

La seule caractéristique essentielle d'un aspect, en tant que module, est de regrouper et d'associer coupes et actions. L'intégration d'un module d'aspect aux autres modules dépend du langage.

AspectJ traite ses aspects comme des classes normales avec une syntaxe étendue. En particulier, il est possible pour un aspect d'hériter d'une classe Java ordinaire (la réciproque est fausse), de déclarer des champs (ligne 17), de définir des méthodes... Le typage nominal permet aussi de déclarer et manipuler la référence d'une instance d'aspect dans une classe Java ordinaire.

AspectJ dispose aussi d'un système élémentaire d'abstraction et de réutilisation à la manière des classes abstraites : un aspect déclaré abstrait peut contenir des coupes et des méthodes

abstraites. Il ne sera pas appliqué au code. Un aspect concret peut étendre un aspect abstrait (mais pas un aspect concret) en définissant les éléments manquants.

3.1.3.5 Tissage

Le tissage désigne le mécanisme de composition modulaire des aspects. Ce mécanisme assemble modules ordinaires et modules d'aspect en tissant aux points de jonction désignés par les coupes les appels aux actions correspondantes.

Outre sa nature implémentatoire, le tissage influence aussi les cas d'utilisation du langage à aspects par son mode d'application. Un tissage à la compilation nécessite l'accès au code source. Un tissage à la liaison d'un programme nécessite un accès au code compilé et est intéressant dans le cadre de bibliothèques d'aspects réutilisables. Un tissage à l'exécution permet de brancher et débrancher dynamiquement un aspect, permettant l'adaptation à la volée tout en assurant une continuité de service.

AspectJ offre deux de ces modes : le mode principal est le tissage à la compilation qui fonctionne dans un processus intégré à la compilation Java standard. Le tissage sur du code Java compilé, au chargement d'un programme, est aussi possible avec des aspects précompilés.

3.1.4 Aspects comportementaux et aspects structuraux

Les caractéristiques d'un langage à aspects sont étroitement liées entre elles : le langage de coupe et le langage d'action sont adaptés au modèle de points de jonction. Ce triplet définit les possibilités et les limites offertes par un langage à aspects. En particulier le couple coupe–action est l'atome manipulé par le programmeur dans la définition d'un aspect.

Au delà des spécificités des coupes et des actions, les mécanismes des langages à aspects peuvent être classés dans deux domaines. Le *domaine comportemental** concerne la modification d'un comportement existant. Le *domaine structurel** concerne la définition d'un nouvel élément.

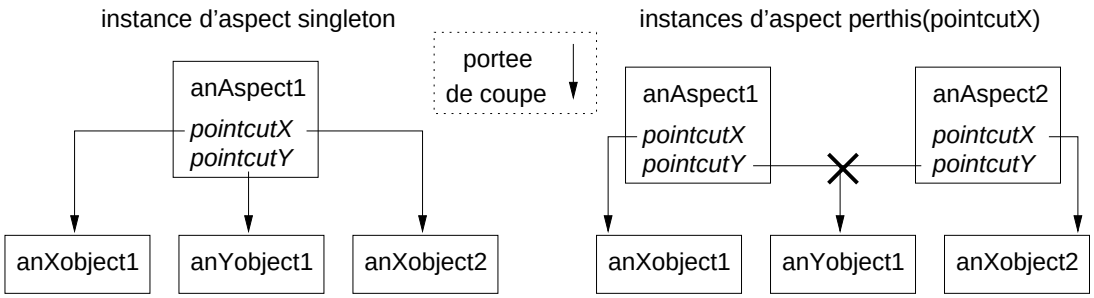
Un aspect est dit comportemental (resp. structurel) si au moins un couple coupe–action concerne ce domaine respectif. Un aspect peut mixer les deux domaines si le langage le permet. Nous exposons ci-dessous les possibilités et les spécificités de quelques langages. Pour le lecteur intéressé, [Filman *et al.*, 2005] décrit de nombreux langages et technologies des aspects.

3.1.4.1 Domaine comportemental

Les caractéristiques d'**AspectJ** présentées en section 3.1.3 concernent le domaine comportemental. Deux autres mécanismes caractérisent de façon importante AspectJ : son modèle d'instantiation et de portée, ainsi que sa gestion des interactions entre aspects.

Comme mentionné en section 3.1.3.4, un aspect est assimilé à une classe Java : il crée donc des instances à l'exécution pour agir. Mais contrairement à une instance de classe, le programmeur n'a pas un contrôle arbitraire sur l'instantiation d'un aspect. Celle-ci est contrôlée par différents modes, dont un seul est choisi par déclaration d'un modifieur dans l'entête de l'aspect : `singleton`, `perthis(coupeInstance)`... Le mode par défaut, `singleton`, est explicite : l'instance se confond avec l'aspect. Les autres modes, comme `perthis(coupe)`, sont paramétrés par une coupe : l'instance d'aspect est associée à une caractéristique de cette coupe. Par exemple, `perthis(coupeInstance)` indique qu'une instance d'aspect sera créée et associée à chaque instance capturée par `coupeInstance`.

Illustration 3.1 – Deux modes d'instantiation et de portée dans AspectJ : `singleton` et `perthis()` (par objet)



Le mode d'instantiation est d'autant plus important qu'il fixe la portée de l'instance d'aspect (figure 3.1). Un aspect singleton peut couper sur l'ensemble des classes (des instances) du programme. Par contre une instance d'aspect par objet (`perthis(coupeInstance)`) ne s'exécutera que sur les coupes capturant l'objet associé ; cette instance d'aspect ne s'exécutera pas sur tout autre objet capturé par d'autres coupes de l'aspect.

Le modèle d'instantiation-portée d'AspectJ a pour but de rendre transparente l'association objet-aspect. La liaison dynamique dans le contexte de la coupe à l'instance d'aspect est déterminée automatiquement. Ce modèle élimine les interactions entre instances d'un même aspect au même point de jonction.

Les interactions entre différents aspects sont liées à la capture du même point de jonction par des coupes : elles sont détectées comme des conflits. AspectJ propose un mécanisme statique de précédence qui définit la priorité entre deux aspects, c'est-à-dire l'ordre dans lequel ceux-ci s'exécutent. Cette déclaration se fait, comme pour le mode d'instantiation, à l'échelle de l'aspect. Il n'est pas possible de déclarer une précédence locale à une coupe.

3.1.4.2 Domaine structurel

AspectJ dispose aussi du mécanisme des *déclarations intertypes* (DIT) pour la définition d'aspects structurels. Ces déclarations permettent de définir dans des classes de nouveaux membres (champs, méthodes – exemple 3.2) ou encore de changer la super-classe ou les interfaces. Un aspect structurel peut donc définir dans le même module des éléments dispersés dans plusieurs classes.

Exemple 3.2 – Exemple de déclaration intertype dans le langage AspectJ. L'aspect `CounterDecr` ajoute une méthode `decr(int)` à la classe `Counter` (exemple 3.1)

```
1 public aspect CounterDecr {
2   // declaration intertype sur Counter
3   public Counter.decr(int delta){
4     this.setValue(value-delta); }
5 }
```

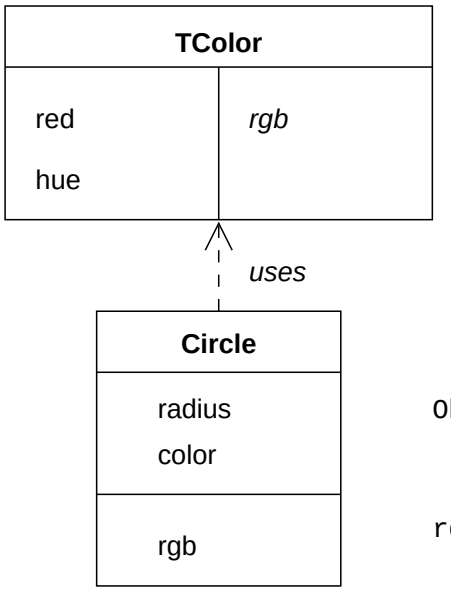
Une application particulière de ce mécanisme est la réutilisation. La déclaration intertype d'un membre peut être faite en liaison avec une interface Java : toute classe implémentant cette interface est augmentée avec ce membre. Cette utilisation des interfaces, transversales à l'arbre d'héritage simple, permet de modifier plusieurs classes par une seule déclaration dans un aspect.

Les déclarations intertypes sont donc proches des mécanismes de réutilisation comme l'héritage multiple, les mixins ou les traits. Nous avons réalisé une comparaison approfondie (voir section 6.4 dans le chapitre 6) des traits avec le mécanisme d'AspectJ pour deux raisons :

- les deux modèles partagent une propriété de composition « plate », c'est-à-dire sans encapsulation des éléments ajoutés ;
- le modèle des traits propose en complément une gestion fine, à l'échelle des méthodes, des conflits de composition – ce mode de résolution est particulièrement intéressant.

Nous présentons plus en détail les caractéristiques du modèle des traits pages 38 et 39.

Illustration 3.2 – Représentation d'un trait `TColor` (d'après Schärli *et al.*). Les méthodes du cadre gauche sont fournies ; les méthodes du cadre droit sont requises. La classe `Circle` est augmentée avec les méthodes fournies de `TColor` en utilisant (`uses`) le trait et définissant la méthode requise (et l'état associé). Le code correspondant est donné en Smalltalk




```
Trait named: #TColor uses:{}

red
     ^ (Color rgb: self rgb) red

hue
     ^ (Color rgb: self rgb) hue

rgb
     self requirement
```

```
Object subclass: #Circle
    uses: {TColor}
    instanceVariableNames: 'radius color'

rgb
     ^ color
```

La programmation adaptive [Lieberherr *et al.*, 2001] est une approche prônant un couplage souple entre structure du programme (ses classes) et comportement (ses méthodes). Ces méthodes dites adaptives définissent des comportements non par rapport aux classes et objets mais par rapport aux relations de ceux-ci : les méthodes « coupent » à travers le graphe des relations inter-objets grâce à des stratégies de traversée.

Les stratégies, assimilées aux coupes, spécifient le parcours dans le graphe de façon abstraite : elles s'adaptent aux changements dans les relations structurelles sans modifier l'implémentation des comportements.

Modèle des traits

Un trait est une brique de réutilisation, sans état, définissant un ensemble de méthodes [Schärli *et al.*, 2003]. Un trait est paramétré par des méthodes requises (ou abstraites). Celles-ci représentent les attributs que le trait ne peut définir. La figure 3.2 montre la vue schématique d'un trait et sa définition en Smalltalk.

Composition des traits. Une classe peut être composée avec un ou plusieurs traits dans sa définition. La classe doit aussi définir (ou hériter) les méthodes requises par le(s) trait(s). La composition est « plate » : une méthode des traits se comporte comme si elle était directement définie dans la classe (en particulier elle redéfinit la méthode homonyme héritée de la superclasse, si celle-ci existe). Un trait composite peut être définie par la composition de plusieurs traits, puis réutilisé dans une classe.

Deux opérations de composition sont utilisées : la somme et la redéfinition.

$$\textit{class } C \textit{ extends } S \ \underbrace{\textit{uses } (\overbrace{T1 + T2 + \cdots}^{\text{Somme}})}_{\text{Redéfinition}}$$

La *somme* de traits est une opération commutative faisant l'union des dictionnaires de méthodes des traits. Elle est utilisée pour la composition d'un ensemble de traits, avant la composition dans une classe ou un trait composite. La somme est une composition symétrique : si deux méthodes de même signature et d'origine différente existent dans l'union, un conflit est déclaré.

La *redéfinition* est la composition d'un trait (ou d'une somme) dans une classe ou un trait composite. Par défaut les méthodes du trait sont intégrées à l'entité composite (classe ou trait). Cette composition est asymétrique : une méthode directement définie dans l'entité composite redéfinit une méthode homonyme du trait utilisé. La méthode issue du trait est donc exclue et il n'y a pas de conflit.

Résolution des conflits. Deux opérateurs supplémentaires permettent d'affiner la résolution de conflits : l'exclusion et l'alias. L'exclusion supprime sélectivement une méthode d'un trait. L'alias crée un accès sous un autre nom à une méthode.

$$\textit{class } C \textit{ extends } S \textit{ uses } (\ \overbrace{T1 - a}^{\text{Exclusion}}\ + \underbrace{T2\,[b := a]}_{\text{Alias}} + \cdots)$$

La redéfinition permet de résoudre les conflits issus d'une somme en remplaçant les méthodes concernées. Un conflit peut être simplement résolu par exclusion des méthodes concernées. La redéfinition et l'alias permettent de fusionner des méthodes en conflit : la redéfinition peut faire appel aux méthodes exclues via leurs alias.

Le modèle des points de jonction est le graphe des relations entre classes et objets. Chaque objet est un nœud et chaque relation une liaison. Un point de jonction est déclenché par la traversée d'un objet dans le graphe. Il faut noter que ce graphe n'est pas nécessairement statique : les dernières incarnations de la programmation adaptive utilisent les relations d'héritage, d'agrégation ainsi que les associations dynamiques issues d'un passage de paramètre ou d'un retour de méthode pour construire le graphe d'un programme.

Le comportement d'une méthode adaptive est défini dans un langage généraliste. La définition de ce comportement est similaire à celle obtenue dans un motif VISITOR.

3.1.5 Notions d'interface transversale

Les techniques de programmation modulaire sont fondées sur la distinction entre interface et implémentation, l'encapsulation de l'implémentation et la notion d'interface en tant que contrat du module. Les bonnes pratiques du génie logiciel accompagnent ces techniques. Le critère de *masquage de l'information* dans le processus de décomposition modulaire [Parnas, 1972] est l'exemple essentiel de ces bonnes pratiques, qui ne peuvent être complètement formalisées dans les langages de programmation mais influencent leur usage.

La jeunesse relative de la programmation par aspects fait que les bonnes pratiques conséquentes ne sont pas encore établies. En premier lieu le concept d'interface transversale est encore largement débattu.

Principe de transparence et problème de modularité. En effet un des premiers principes soutenant la programmation par aspects est la transparence [Filman et Friedman, 2000]. Suivant ce principe, un programme peut être développé a priori sans tenir compte des préoccupations transversales. Les aspects sont ajoutés a posteriori de façon transparente, sans modification du code de base. Cette démarche repose sur deux hypothèses : d'une part le langage d'aspects et en particulier le langage de coupes doivent être suffisamment expressifs pour exploiter la trace du programme ; d'autre part le tisseur doit avoir un accès complet au code afin d'assurer l'intégralité des modifications sémantiques décrites par les aspects.

Les travaux actuels montrent que la première hypothèse est difficile à assurer :
[Gybels et Brichau, 2003] indique que de nombreux programmes doivent être modifiés pour accomoder les coupes d'AspectJ. De plus celles-ci sont souvent fragiles car trop sensibles à la syntaxe. La seconde hypothèse implique que l'aspect ne s'arrête pas aux interfaces des modules : il peut accéder à n'importe quel détail de l'implémentation, ce qui est une violation de l'encapsulation modulaire.

Travaux exploratoires. Plusieurs travaux explorent donc le concept d'interface transversale afin d'assurer un contrat modulaire.

[Aldrich, 2005] propose les *Open Modules*. Ce système de modules rétablit l'encapsulation et intègre les coupes à l'interface du module visé. Ces coupes capturent les points de jonction internes au module et sont exposées dans l'interface. Un aspect extérieur a accès aux éléments de cette interface –appel des méthodes et des coupes déclarées– mais pas aux points de jonction non exposés. Le programmeur du module visé par l'aspect doit donc garantir la définition et la maintenance des coupes de l'interface, ce qui est en opposition avec le principe de transparence. Cependant, si un aspect est intéressé par plusieurs modules, chacun de ces modules devra par construction déclarer la coupe adéquate : on observe donc une forme (plus légère) de dispersion.

Une autre approche consiste à développer des langages de coupe de haut niveau. [De Volder et D'Hondt, 1999] utilise la programmation logique afin de déclarer des coupes qui, à partir de règles logiques, déduisent les points de jonction adéquats. [Ostermann *et al.*, 2005] complémente la programmation logique avec différents modèles du langage et de sa sémantique, afin d'enrichir la représentation des points de jonction. Cette approche cherche à définir différents niveaux d'abstraction adéquats pour raisonner sur le code source, satisfaisant la première hypothèse du principe de transparence : la finalité est que les interfaces transversales, déclarées par les coupes, soient intégrées aux aspects les utilisant sans rien révéler de l'implémentation des modules ciblés.

Cette opposition entre transparence et encapsulation se retrouve enfin dans les méthodes et principes discutés dans [Kiczales et Mezini, 2005] et [Griswold *et al.*, 2006].

[Kiczales et Mezini, 2005] redéfinissent le concept d'interface en présence d'aspects avec le concept d'interfaces aspectisées (*aspect-aware interfaces*). Ces interfaces sont produites par le tisseur et contiennent, pour un module, son interface déclarée ainsi que les coupes imposées par les aspects. Suivant ce principe, les aspects cassent la définition localisée de l'interface par le programmeur. De plus, le calcul d'interfaces n'est valide que pour une configuration donnée, c'est-à-dire un ensemble fixé de modules, car le tisseur doit connaître l'intégralité des aspects et des modules à composer. Cependant ce calcul global permet ensuite le raisonnement modulaire pour la configuration donnée puisque chaque interface expose les interactions avec les autres modules. Dans ce cadre le principe de transparence est défendu afin de permettre l'expression optimale des aspects.

[Griswold *et al.*, 2006] propose une méthode de conception des interfaces transversales, appelées XPI (*Crosscut Programming Interfaces*, où X se lit *crosscut*). Cette méthode prône comme principe la séparation entre interface transversale à un module et aspect faisant usage de cette interface pour implémenter une préoccupation transversale. À la manière des *Open Modules*, l'interface transversale abstrait les détails d'implémentation du module dans la déclaration des coupes adéquates et documente les contrats que doivent respecter ces coupes. Cette méthode, présentée avec AspectJ, n'offre donc pas une garantie forte sur l'encapsulation : l'aspect doit par convention n'utiliser que les coupes de l'interface transversale (extraite de l'aspect) pour adapter un module. La maintenance de l'interface transversale, représentée par les coupes, est à la charge du programmeur du module visé. En particulier rien n'interdit à celui-ci de restructurer son module pour accommoder les coupes. Bien que cette méthode fasse un usage non restreint du pouvoir d'expression d'AspectJ, elle ne suit donc pas le principe de transparence.

3.2 Approche des motifs avec la programmation par aspects

La programmation par aspects vise à améliorer la modularité des programmes. Il est donc intéressant d'observer comment les problèmes de modularité des motifs (section 2.2.2) peuvent être résolus. Les langages d'aspects permettent de capturer et d'encapsuler les éléments d'implémentation dispersés des motifs.

Cependant, comme souligné en section 2.2.1.1, les patrons et encore plus les motifs sont relatifs aux mécanismes des langages de programmation. L'utilisation des nouveaux mécanismes des aspects transforment donc aussi les motifs.

Plus précisément, la démarche des aspects est intéressante car elle permet *l'inversion des dépendances* [Nordberg III, 2001a]. Or [Nordberg III, 2001b] souligne que la majorité des motifs

fonctionnent par une indirection entre différents rôles. Ces indirections impliquent des dépendances ainsi que des implémentations invasives et dispersées (sections 2.2.2.1 et 2.2.2.2).

L'aspectisation d'un motif est la conjugaison de deux axes : transformation de certains éléments de l'implémentation et capture des éléments non transformés pour la modularisation de l'ensemble. Par la suite nous examinons plusieurs mécanismes transformant certains motifs, étudions la modularisation du motif Observer par différents langages à aspects et présentons les études extensives réalisées sur les motifs avec les aspects.

3.2.1 Transformation des motifs

L'axe fondamental dans l'approche des motifs avec la programmation par aspects est la transformation (d'une partie) de la solution objet de ces motifs par la prise en compte des mécanismes aspects. Ainsi dans le motif Observer, les appels de méthodes déclenchant les notifications sont éliminés grâce aux coupes AspectJ, lesquelles ciblent les points de jonction correspondant (voir exemple 3.1). La section suivante détaille diverses approches pour modulariser l'intégralité du motif Observer, toutes basées cependant sur la transformation des notifications par les coupes.

[Isberg, 2005] est une expérimentation technique examinant la conception de JUnit [Gamma et Beck, 2002] avec la programmation par aspects. Les problèmes exposés par les patrons sont revisités avec les mécanismes d'AspectJ. L'exercice pousse au bout la logique de transformation par les aspects puisque les motifs sont ignorés. La démarche n'indique pas cependant la récupération de l'expertise par de nouveaux motifs.

La programmation adaptive [Lieberherr *et al.*, 2001] (voir aussi page 38) permet la séparation du comportement et de la structure à la manière du patron Visitor. Il est ainsi possible de définir un nouveau comportement sans modifier les classes de la structure. La solution des stratégies et des méthodes adaptives améliore cependant le motif Visitor sur plusieurs points : les stratégies prennent en charge la traversée de structures arbitrairement complexes ; les stratégies sont abstraites donc plus souples par rapport aux changements de structure – il est possible de modifier celle-ci sans impact sur les méthodes adaptives, contrairement au motif Visitor ; enfin le surcoût d'implémentation lié aux indirections dans le motif Visitor disparaît.

Les *Composition Filters* [Bergmans et Akşit, 2001] est une extension du modèle objet dont le mécanisme central est l'interception des messages. Plusieurs filtres peuvent être composés autour d'un objet (ou d'un ensemble d'objets). Ces filtres peuvent intercepter les messages reçus par les objets, les refuser, les rediriger, les modifier. Chaque filtre permet de modifier le comportement d'un objet de façon dynamique, sans changer l'implémentation de la classe. Cette fonctionnalité répond au patron Decorator.

3.2.2 Exemples de modularisation du motif Observer

Cet axe vise à capturer et encapsuler les éléments du motif qui ne sont pas transformés par les mécanismes aspects.

L'implémentation du motif Observer constitue à ce titre un exemple typique :

1. il y a création d'une dépendance du rôle sujet vers le rôle observateur ;
2. il est courant que les notifications soient dispersées sur plusieurs méthodes et plusieurs classes du rôle sujet ;

3. le sujet doit définir et maintenir une liste d'observateurs.

L'implémentation du motif OBSERVER est donc invasive dans le rôle sujet. Ceci (en particulier le point 2) en fait une cible de choix pour la démonstration des mécanismes et des principes des nouveaux langages à aspects.

[Hannemann et Kiczales, 2002] fait la démonstration d'une démarche systématique avec AspectJ : les rôles sont capturés par des interfaces Java, la structure par des tables de relation, les collaborations par des coupes-actions ou des méthodes internes à l'aspect. Cette démarche transforme les collaborations entre rôles et conduit à modulariser tous les éléments d'un motif dans un aspect singleton. Les déclarations intertypes sont aussi utilisées pour étendre les classes ciblées.

Caesar [Mezini et Ostermann, 2003] est une extension de Java et d'AspectJ modifiant en particulier le modèle d'instance d'AspectJ (section 3.1.4.1) pour le rendre plus souple. Caesar permet de créer facilement une instance d'aspect pour chaque objet impliqué dans un rôle. Cette instance modifie le comportement de l'objet en conséquence. Le modèle d'implémentation du motif est alors plus proche du modèle objet classique.

Reflex [Tanter, 2004] est un cadriciel pour la réflexion partielle en Java. Il est conçu comme un noyau réflexif permettant l'implémentation de différents langages à aspects. Il propose des concepts et des primitives plus basiques mais aussi un modèle plus général qu'AspectJ. Le concept de lien entre un méta-objet (assimilé à une instance d'aspect) et un ensemble de crochets (*hookset*, correspondant à un ensemble de points de jonction) facilite l'association d'un aspect à un ensemble arbitraire d'objets : le rôle observateur du motif OBSERVER est par exemple capturé dans un méta-objet, sans que l'implémentation du sujet soit impactée [Tanter *et al.*, 2003].

[Clarke et Walker, 2001] propose les *Composition Patterns*, basée sur la modélisation des motifs par des diagrammes UML à stéréotypes, puis sur leur transposition en AspectJ. L'objectif de cette démarche est de constituer des modèles réutilisables des motifs. Il faut noter que le terme de composition traduit l'application de ces modèles et non la composition de motifs exposée en section 2.2.3.

3.2.3 Études extensives sur l'aspectisation des motifs du *GoF*

Nous citons ici les travaux utilisant les patrons du *GoF* afin de valider les concepts et les mécanismes des langages à aspects. L'objectif de ces travaux, tous réalisés avec AspectJ, est avant tout d'évaluer le comportement du langage vis-à-vis des problèmes objet représentés par ces patrons.

3.2.3.1 Étude qualitative

[Hannemann et Kiczales, 2002] est la seule étude extensive qualitative à ce jour sur l'aspectisation des motifs. La démarche de transformation et de modularisation est appliquée de façon systématique.

Selon Hannemann *et al.*, la transformation par AspectJ montre pour la plupart des motifs un résultat bénéfique en terme de localité du code, réutilisation d'une version générique, trans-

parence de composition[5] et branchement modulaire (*pluggability*). Ce travail révèle aussi une distinction plus fine des rôles des motifs. Ceux-ci se classent en deux catégories :

- un rôle *surimposé* vient s'ajouter à la préoccupation principale de la classe. Il se mélange avec celle-ci et fait donc un aspect intéressant.
- un rôle de *définition* constitue la préoccupation principale de la classe. La classe définit un comportement spécifique pour ce rôle, délicat à extraire.

Cette indication permet d'opérer un tri entre les motifs mais aussi au sein des motifs : certains motifs et certains rôles peuvent être aspectisés tandis que d'autres garderont une conception objet classique.

Cependant cette démarche systématique révèle aussi les idiomes de programmation du langage AspectJ. L'impact de ces idiomes sur l'implémentation des motifs et leurs spécificités vis-à-vis d'AspectJ ne sont pas évalués.

3.2.3.2 Études quantitatives

[Garcia *et al.*, 2005] réalisent une étude quantitative des implémentations de motifs proposées par Hannemann *et al.*. Cette étude évalue la qualité des implémentations avec AspectJ grâce à une série de métriques objets adaptées aux aspects. Elle confirme les résultats généraux de l'étude tout en nuançant certaines propriétés : en effet les solutions présentées permettent une meilleure séparation mais augmentent parfois la taille du code, la complexité des opérations ainsi que le couplage.

[Cacho *et al.*, 2006] étend le travail précédent avec une étude de compositions de motifs grâce aux aspects. Le but de cette étude est avant tout d'évaluer par des métriques la qualité des programmes transformés avec AspectJ, et non l'implémentation de ces compositions. La typologie suivante des compositions est cependant proposée.

Invocation : la composition implique l'appel de méthodes entre motifs mais les implémentations ne concernent pas les mêmes classes.

Entrelacement intraclasse : les implémentations des motifs ciblent les mêmes classes sans mélanger leurs éléments respectifs.

Entrelacement intraméthode : les implémentations ciblent les mêmes méthodes, en conséquence de quoi leurs comportements sont mélangés dans ces méthodes.

Recouvrement : les implémentations partagent et utilisent les mêmes éléments. Il n'y a pas de mélange mais utilisation conjointe.

L'étude relève, sans donner de détail, que certaines compositions ne sont pas modulaires : la composition de deux motifs peut demander un raisonnement global, non modulaire, et amener à des transformations du code de base.

3.3 Conclusion

La programmation par aspects est une technique complémentaire des objets. Elle vise la modularisation des préoccupations transversales des programmes. Ces préoccupations se manifestent par la dispersion et le mélange des implémentations dans les décompositions modulaires

[5] La transparence de composition indique que plusieurs occurrences du même motif peuvent cohabiter sans interférer.

hiérarchiques. La programmation par aspects est fondée sur les concepts de point de jonction (événement du programme), de coupe (désignation d'un ensemble de points de jonction) et d'action à exécuter au déclenchement de ces coupes. Un aspect est, du point de vue conception et implémentation, un module encapsulant coupes et actions associées à une préoccupation transversale.

L'implémentation des motifs de conception, outre les problèmes de modularité, est parfois associée aux symptômes de dispersion et de mélange des préoccupations transversales. Ceci fait des motifs des candidats intéressants au titre d'aspect. La démarche d'aspectisation d'un motif suit deux axes conjugués : transformation par les mécanismes des langages à aspects et modularisation dans un aspect. Les travaux ayant exploré de façon extensive l'aspectisation des motifs standards indiquent globalement un processus bénéfique, bien que leurs conclusions ne se recouvrent pas complètement.

Partie II

Contributions

Sommaire

À partir des éléments de notre état de l'art, nous résumons la problématique de la densité, de l'implémentation et de la composition des motifs. Nous présentons nos quatre thèmes de contributions et leur articulation : l'étude de la densité des motifs, l'implémentation et la composition des motifs avec les aspects, la complémentarité des aspects et des objets en Java avec AspectJ.

Densité des motifs de conception

Les motifs de conception sont considérés comme des bonnes pratiques de la programmation par objets. Leur mise en œuvre dans les programmes facilite l'adaptation de ceux-ci. En effet les patrons de conception présentent un « aspect de variabilité » selon les termes du [Gamma *et al.*, section 1.7, tableau 1.2]. Les cadriciels font usage des motifs pour définir leurs adaptations grâce à cette variabilité. La forte densité de motifs observée dans les cadriciels (section 2.2.3.1 page 21) est un indice de cet usage.

Mais l'absence de traçabilité, les interdépendances, le manque de réutilisation soulignent les problèmes de modularité liés à l'implémentation des motifs avec les langages à objets (section 2.2.2 page 18). Plus précisément, l'implémentation d'un motif crée de nouvelles dépendances entre classes, ce qui entraîne des modifications invasives au détriment de la modularité des classes (section 2.2.2.2 page 19 et section 3.2 page 41).

Les motifs entraînent donc des « micro » violations de modularité, à l'échelle d'une classe, dans le but d'obtenir une meilleure « macro » modularité, liée à l'adaptation à l'échelle d'un ensemble de classes. Or en l'absence d'implémentation modulaire, il est difficile d'étudier et prévoir l'impact des motifs et de leur composition sur le code. Les motifs sont seulement juxtaposés et spécialisés de façon ad hoc dans le code (section 2.2.3 page 21). Il est d'autant plus difficile de comprendre et maîtriser les effets d'une forte densité, signe de l'implémentation et de la composition de multiples motifs.

Ce constat est aussi lié au caractère informel de la description des patrons en langage naturel et à la disparité des motifs de conception (section 2.2.1.5), qui rendent difficile toute généralisation à partir d'un modèle. Pour cette raison, malgré divers travaux de classification et de formalisation [Baroni *et al.*, 2003], nous ne connaissons pas de modèle permettant la prévision des résultats d'une composition.

Ces divers éléments nous amènent dans le chapitre 4 à une étude de cas sur le cadriciel JHotDraw, afin de clarifier les conséquences concrètes d'une forte densité.

Motifs de conception et programmation par aspects

Plusieurs approches langages ont été proposées pour supporter l'implémentation des motifs (section 2.3.1 page 25). Cependant la disparité des motifs rend difficile toute proposition générique ; de même la relativité d'un motif indique qu'un mécanisme langage peut être adéquat pour un motif et non pour un autre (section 2.2.1.1). En étudiant les motifs objets avec la programmation par aspects, notre objectif général n'est pas de transformer tous les motifs mais de déterminer quels motifs sont les mieux supportés par les aspects.

Implémentation des motifs avec les aspects

Les symptômes de dispersion et de mélange des implémentations des motifs en font des bons candidats à leur modularisation par les aspects (section 3.2 page 41). Cependant l'implémentation des motifs avec les aspects passe par la transformation de leurs solutions pour exploiter les nouveaux mécanismes des aspects (section 3.2.1). Pour certains motifs, cette transformation est complète : le motif aspectisé est une nouvelle solution au problème exposé par le patron.

Si la transformation n'est que partielle, la modularisation passe aussi par la définition locale (dans l'aspect) des éléments normalement dispersés de l'implémentation (section 3.2.2). Cette capture des éléments dispersés implique de transcrire une partie de la conception objet du motif dans l'aspect.

Enfin, le fait de transformer les implémentations des motifs change aussi leurs propriétés. Même si un motif aspectisé réalise l'intention du patron, il n'est pas utilisé de la même manière ni n'a la même structure ou le même comportement que son homologue objet. Ces différences peuvent avoir un impact au moment de choisir entre une solution objet et une solution aspect.

Dans le chapitre 5, nous examinons les cas de figure ci-dessus à travers plusieurs examples détaillés ainsi qu'une revue extensive des « aspectisations » (modularisations par les aspects) des motifs standards par [Hannemann et Kiczales, 2002].

Composition des motifs avec les aspects

La modularisation des implémentations des motifs par les aspects permet d'envisager la composition des motifs dans le chapitre 6. La composition peut impliquer deux motifs à des degrés divers de dépendance, en fonction de leurs préoccupations. Nous proposons le concept de *modalité** pour décrire cette composition abstraite entre motifs.

Nous appliquons ce concept pour discriminer différentes solutions de composition de deux motifs. Nous examinons les propriétés et les compromis des compositions en terme de modularité, de réutilisation et d'adaptation.

Enfin nous examinons le traitement des interactions et des conflits entre motifs par les mécanismes des aspects. Il s'agit d'évaluer les solutions des aspects face aux interactions et conflits potentiels liés à une forte densité de motifs.

Méthode de programmation des objets et des aspects

Dans cette thèse nous nous concentrons sur le langage à objets Java et son extension pour les aspects AspectJ. AspectJ est en effet un langage d'aspects suffisamment mature et stable pour être employé de façon industrielle. Il présente à la fois des mécanismes comportementaux et structurels (coupe-action et déclaration intertype, section 3.1.4.1 page 36) qui permettent plusieurs solutions possibles.

Puisqu'AspectJ produit un code compilé compatible avec Java, il est possible de voir les aspects comme des objets normaux. La programmation avec AspectJ se fait donc de façon à la fois conjointe et complémentaire à Java. Au chapitre 7, nous capitalisons les résultats obtenus avec l'aspectisation et la composition des motifs de conception dans une méthode de programmation pour AspectJ. Cette méthode s'applique en particulier à la restructuration d'un

programme objet par les aspects. L'application de cette méthode à JHotDraw nous permet de résoudre les problèmes découverts au chapitre 4.

Chapitre 4

Étude de cas sur la densité des motifs dans JHotDraw

Sommaire

L'étude du cadriciel JHotDraw illustre le pouvoir d'abstraction et l'impact sur l'implémentation lié à la densité des motifs de conception.

Notre démarche se base sur un noyau de JHotDraw : un ensemble réduit d'interfaces et de classes capturant les préoccupations essentielles du cadriciel. Nous montrons comment les motifs présents dans le noyau permettent d'expliquer de façon synthétique les collaborations internes et les extensions principales du cadriciel. Cette densité est éclairée par deux métriques.

Par la suite de nouvelles extensions, basées sur l'ajout ou le retrait de motifs, modifient la densité dans le noyau. Des effets positifs et négatifs sont observés sur l'implémentation : des opportunités pour la simplification et la réutilisation sont ouvertes ; mais ces changements impliquent aussi des modifications invasives et la dispersion des implémentations.

4.1 Introduction

La densité des motifs est une notion encore floue (section 2.2.3.1). Faute de modèle permettant l'étude a priori (voir page 48), nous procédons dans ce chapitre à l'analyse a posteriori d'un programme à forte densité afin d'éclairer les effets possibles.

Intuitivement, la densité est un rapport entre le nombre de motifs implémentés dans le code et le nombre de classes. En effet un motif implique généralement plusieurs classes et une classe peut se trouver impliquée dans plusieurs motifs. Pour un même ensemble de classes, plus le nombre de motifs impliqués sera grand, plus la densité sera forte et plus les motifs « supportent » les classes.

Nous procédons à une analyse à deux niveaux : un niveau conceptuel et un niveau implémentation. Au niveau conceptuel, nous utilisons les motifs pour décrire et expliquer le fonctionnement interne du programme. Dans le cadre d'une forte densité, nous montrons comment les motifs permettent une documentation synthétique du code, en particulier de ses collaborations et extensions. Nous définissons aussi deux métriques de densité servant d'indicateurs pour ce pouvoir d'abstraction.

Au niveau implémentation, nous observons l'impact sur le code lié à l'ajout (ou au retrait) de motifs. Faute de modularité, cet impact est invasif (insertion de code dans la classe visée par le motif) mais nous montrons qu'il peut aussi être transversal (modification d'une classe `A` suite à l'implémentation d'un motif dans une classe `B`). Nous montrons enfin comment une forte densité offre des opportunités pour la réutilisation du code des motifs, au risque cependant de fragiliser la modularité des implémentations.

La densité traduit la présence de motifs mais ne permet pas de décrire leurs compositions. Celles-ci se résument dans ce chapitre à juxtaposer, adapter et réutiliser si possible les implémentations. En définitive nous exploitons la densité comme un indicateur abstrait de la maturité du code et non comme une mesure concrète de sa complexité.

Sujet de l'étude : JHotDraw

Pour cette étude nous avons choisi JHotDraw car il s'agit d'un exemple concret et non trivial d'application, tout en restant d'une taille raisonnable et facilement maîtrisable. JHotDraw est un cadriciel pour applications de dessin structuré (dessin vectoriel et technique). Ce cadriciel a en particulier été conçu comme banc de démonstration des patrons de conception avec trois objectifs : aide à la conception et à l'implémentation, documentation de la conception et du code, utilisation (extension) du cadriciel. Les motifs utilisés sont indiqués dans la documentation de conception et dans les commentaires du code.

Pour simplifier l'étude nous avons établi un noyau de JHotDraw et déterminé quels motifs s'y trouvaient imbriqués (et dans quel but). Nous avons ensuite expérimenté l'extension du noyau par de nouvelles préoccupations impliquant des motifs supplémentaires.

4.2 Étude par les motifs du noyau de JHotDraw

JHotDraw se présente comme un cadriciel pour application *WYSIWYG* (*What You See Is What You Get*) de dessin vectoriel. L'utilisateur final de l'application (le dessinateur) dispose d'une fenêtre affichant son dessin et permettant la manipulation directe de figures, ainsi que

des moyens (menus de commandes, barre d'outils) d'interagir avec ces figures. Les figures représentent les objets du domaine de l'application. Cette approche est d'autant plus intuitive que le domaine du dessin technique s'y prête de façon évidente.

Pour implémenter une nouvelle application, l'utilisateur du cadriciel (le programmeur) a la charge de définir les classes du domaine et d'étendre les classes nécessaires du cadriciel. On retrouve au cœur du cadriciel les principes d'une architecture MVC (Modèle-Vue-Contrôleur [Reenskaug, 1979]) : séparation entre le modèle des données, leur affichage et les contrôles permettant de modifier ces données. Cependant la frontière entre modèle et affichage est ici floue du fait du caractère éminemment graphique des données de base, qui sont des figures. Outre les figures, on retrouve le concept de vue ainsi que différents moyens de contrôle. Ces catégories partagent les entités du noyau présenté ci-dessous.

4.2.1 Définition et principes du noyau

Le point de départ de cette étude est l'isolation d'un sous-ensemble de classes et d'interfaces du cadriciel original constituant un noyau du cadriciel.

> Définition 4.1 – Noyau du cadriciel
>
> Le noyau est le plus petit sous-ensemble de classes et d'interfaces nécessaires à toute application de JHotDraw. Il définit les collaborations internes ainsi que les points d'extension essentiels.

Ainsi des éléments comme la vue, la figure ou l'outil sont essentiels au cadriciel et à son extension. D'autres préoccupations, comme la gestion d'un historique de travail (*undo*), sont considérées comme accessoires et exclues du noyau. Elles sont qualifiées d'extensions externes par opposition aux extensions essentielles internalisées par le noyau.

Nous encadrons la définition du noyau avec trois principes :

principe d'économie : l'implémentation est simplifiée de façon à répondre aux seules spécifications du noyau, sans faire d'hypothèse sur les extensions externes ;

principe d'encapsulation : l'implémentation est fermée aux modifications par des extensions externes ;

principe d'extension modulaire : toute extension, essentielle ou externe, doit être modulaire, que ce soit par rapport au noyau (encapsulation) ou entre elles.

Les principes d'économie et d'encapsulation entraînent la suppression des implémentations jugées accessoires dans les classes et interfaces du noyau. L'objectif est d'obtenir un noyau simple et suffisant pour comprendre et garantir les collaborations et extensions essentielles du cadriciel.

4.2.2 Architecture objet du noyau

Pour décrire et documenter le noyau, nous employons le terme d'*entité** . Ce terme permet de s'abtraire des classes et interfaces Java de l'implémentation.

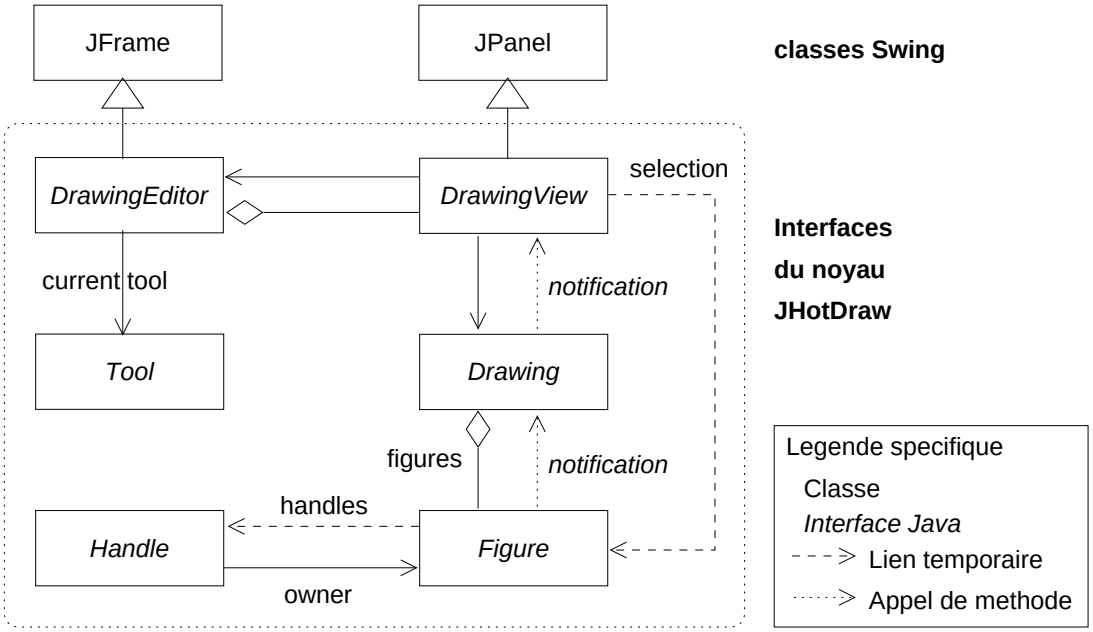
La figure 4.1 illustre de façon synthétique les entités du noyau ainsi que leurs relations. Ce schéma est dérivé du schéma initial fourni dans la documentation de JHotDraw 5.2, avec quelques

Définition 4.2 – Entité du cadriciel

Une entité est un concept essentiel au fonctionnement et à l'extension du cadriciel. Elle se substitue dans la conception aux classes et interfaces de l'implémentation. Elle est représentée, dans l'implémentation de JHotDraw, par une interface Java pour le type et par une classe pour les attributs et le comportement standard.

adaptations mineures reflétant l'état courant dans la version 5.3[6]. La relation d'héritage entre la classe Swing `JFrame` et l'entité JHotDraw `DrawingEditor` est un raccourci pour la relation réelle entre `JFrame` et la classe `DrawApplication`, qui implémente l'interface `DrawingEditor` – de même pour la classe Swing `JPanel` et l'entité `DrawingView`, implémentée par défaut par `StandardDrawingView`. Le noyau de JHotDraw correspond à ces six entités ainsi qu'aux classes annexes utilisées par ces entités (non montrées).

Illustration 4.1 – Schéma synthétique des entités et relations du noyau JHotDraw



Nous décrivons maintenant les responsabilités et les relations des entités du noyau, ainsi que leur extension dans le cadre d'une utilisation du cadriciel.

`DrawingEditor` est l'entité représentant l'application et assurant la liaison entre la vue active (`DrawingView`) du dessin et les outils et commandes de l'interface. L'implémentation standard (et réutilisable) `DrawApplication` hérite de la classe Swing `JFrame`. Elle crée un cadre d'application. Son extension a donc pour but de spécialiser l'interface graphique et d'instantier les outils, vues et modèles spécifiques à l'application.

[6] Accessible à l'adresse `http://www.jhotdraw.org/`.

`DrawingView` gère l'affichage du dessin ainsi que la transmission des événements utilisateurs (souris) aux commandes et une partie des interactions (détection et sélection de figures). Cette entité joue un rôle soit central soit intermédiaire dans la plupart des activités du cadriciel. L'implémentation standard `StandardDrawingView` hérite de la classe Swing `JPanel` pour les tâches d'affichage et la prise en charge des *listeners* Swing. Bien que cette implémentation soit suffisamment complète et générique pour être réutilisée telle quelle, la compréhension de nombreux détails et collaborations passe par une analyse approfondie de cette entité.

`Drawing` contient un ensemble de figures et offre un point d'entrée unique pour leur gestion. Comme pour `DrawingView`, l'implémentation standard `StandardDrawing` peut être réutilisée sans extension particulière.

`Figure` représente l'élément de base affiché et manipulé dans un dessin, avec des actions telles que dessiner, déplacer, changer la taille. C'est donc une abstraction centrale pour l'extension du cadriciel, en relation avec le modèle du domaine d'application (le modèle peut être directement implémenté dans les figures). Suivant l'application dérivée du cadriciel, l'arbre d'héritage dérivant de `Figure` peut être plus ou moins important.

`Tool` représente la catégorie des actions réagissant directement aux événements souris (créer une figure, sélectionner une figure, déplacer une figure). `DrawingEditor` désigne l'outil actif, lequel s'exécute suite aux événements transmis par `DrawingView`. Comme pour `Figure`, son extension est dépendante du domaine d'application.

`Handle` représente une autre catégorie d'actions, manipulant à la souris les propriétés (taille, rayon) de la figure sélectionnée. C'est en fait la figure qui instancie et fournit à la demande les `Handle` spécifiques à sa classe. Contrairement aux outils, ceux-ci sont dépendants de leur figure parente.

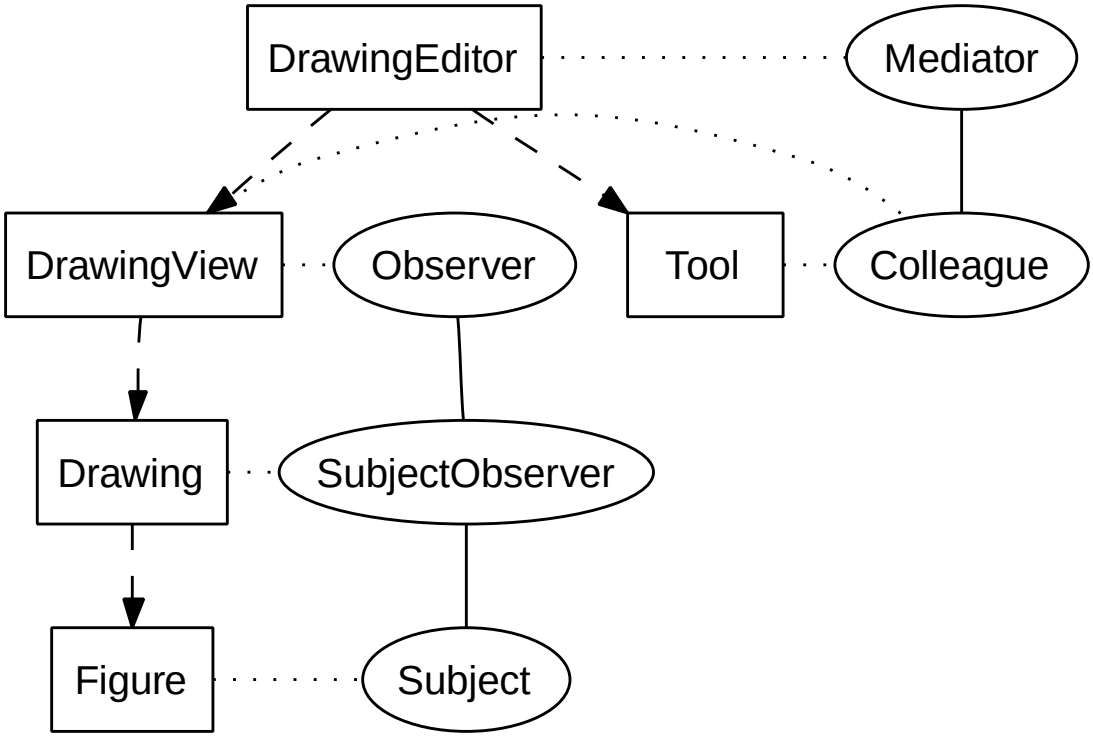
Synthèse La description succincte des entités donne une idée des possibilités fondamentales du cadriciel JHotDraw. Mais cette description reste insuffisante pour comprendre le fonctionnement du cadriciel et entreprendre son extension.

4.2.3 Implication des motifs de conception dans le noyau

Nous allons maintenant utiliser les motifs de conception pour décrire certaines fonctions et collaborations des entités. Cet usage documente le cadriciel, tant sur son fonctionnement que sur son extension, et est complémentaire à la présentation des entités ci-dessus. La connaissance transmise à travers les patrons et les motifs permet à un expert de deviner la solution sans avoir vu le code : cette vue est donc à la fois plus précise que la description naturelle mais reste économe car elle n'est pas encombrée par les détails d'implémentation.

Les motifs que nous présentons sont issus aussi bien de la documentation de conception que des commentaires laissés dans le code. Nous nous sommes cependant intéressés aux seuls motifs directement impliqués dans le noyau. Pour chaque motif, nous rappelons brièvement les éléments importants puis détaillons les *occurrences** dans le noyau.

Illustration 4.2 – Implication des motifs MEDIATOR et OBSERVER dans le noyau de JHotDraw



4.2.3.1 Mediator

Le motif MEDIATOR permet de découpler les participants d'une collaboration complexe en centralisant les interactions. Ainsi les dépendances directes entre participants sont réduites voire éliminées.

Dans le cas de JHotDraw, `DrawingEditor` est un médiateur entre les outils et les vues (figure 4.2). Cependant il ne correspond pas au motif MEDIATOR standard car il n'implémente pas les interactions entre ces entités. Il est simplement utilisé par l'outil courant pour obtenir, de façon dynamique, la référence à la vue courante et réciproquement. Son but est d'encapsuler les processus de changement d'outil et de vue au sein de `DrawingEditor`.

4.2.3.2 Observer

Le motif OBSERVER est courant dans les architectures MVC, où il assure la cohérence des dépendances entre les tierces parties par un mécanisme de notification.

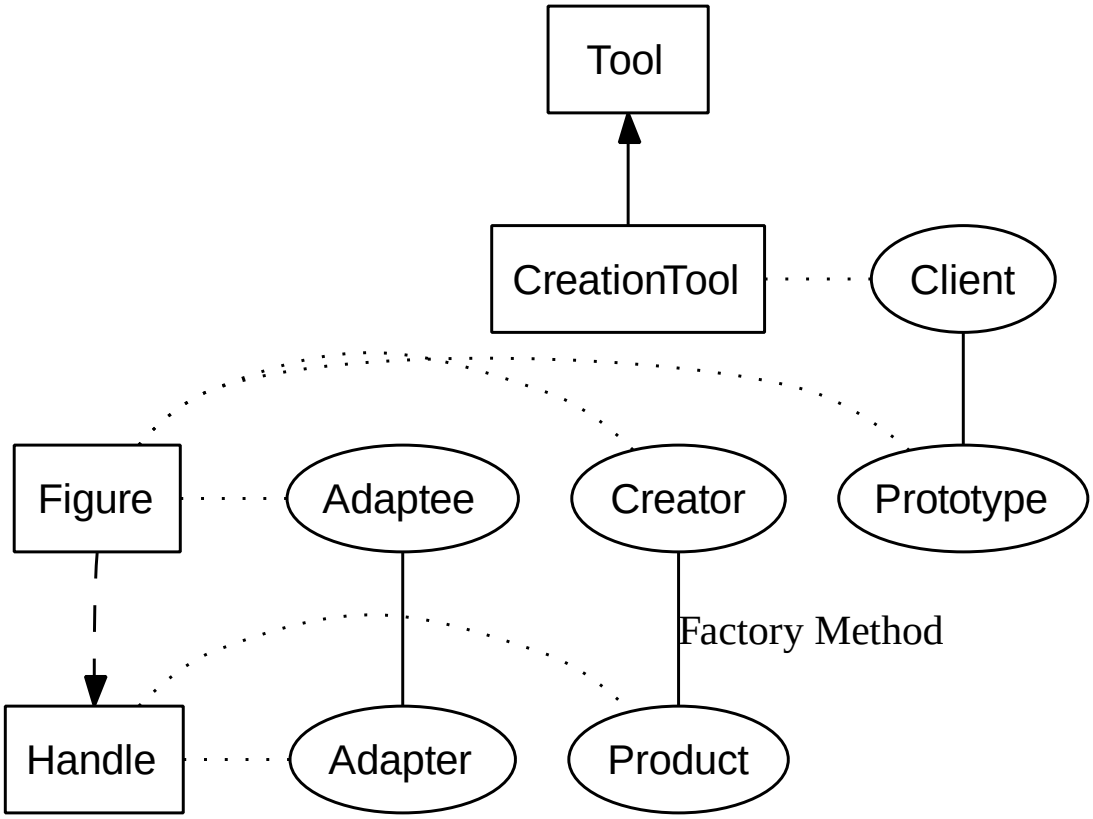
Deux occurrences existent dans le noyau : dans la première le dessin observe ses figures ; dans la seconde, la vue observe son dessin (figure 4.2). Les deux occurrences sont partiellement liées (un événement déclenché par une figure peut se propager à la vue via le dessin). Cette liaison permet la mise à jour de la vue après modification d'une figure. Nous détaillons en section 4.3.1 cette préoccupation et son implémentation. D'autres occurrences sont, par exemple, dédiées aux interactions homme–machine (activation des boutons et menus de l'interface).

4.2.3.3 Strategy

Le motif STRATEGY permet de changer dynamiquement une action en délégant celle-ci à l'objet approprié. C'est une application directe de la délégation, un des mécanismes fondamentaux des langages à objets.

Deux occurrences (non montrées) en particulier existent dans `DrawingView` : la gestion du tampon de dessin (qui peut être à simple ou à double tampons) et la gestion de la matrice du dessin (contraignant la position des points aux intersections d'une grille).

Illustration 4.3 – Implication des motifs autour de `Figure`



4.2.3.4 Prototype

Le motif PROTOTYPE propose une implémentation simple pour la création d'objets de types et de paramètres différents, en clonant des instances typiques.

Son usage se fait dans le cadre de la création des figures (figure 4.3). Un prototype est instancié au démarrage pour chaque type de figure. La création d'une figure se fait par clonage du prototype via l'outil générique `CreationTool`.

4.2.3.5 Adapter

Le motif ADAPTER sert à émuler une interface pour un objet sans modifier son implémentation. ADAPTER redirige les appels de son interface vers l'interface existante de l'objet.

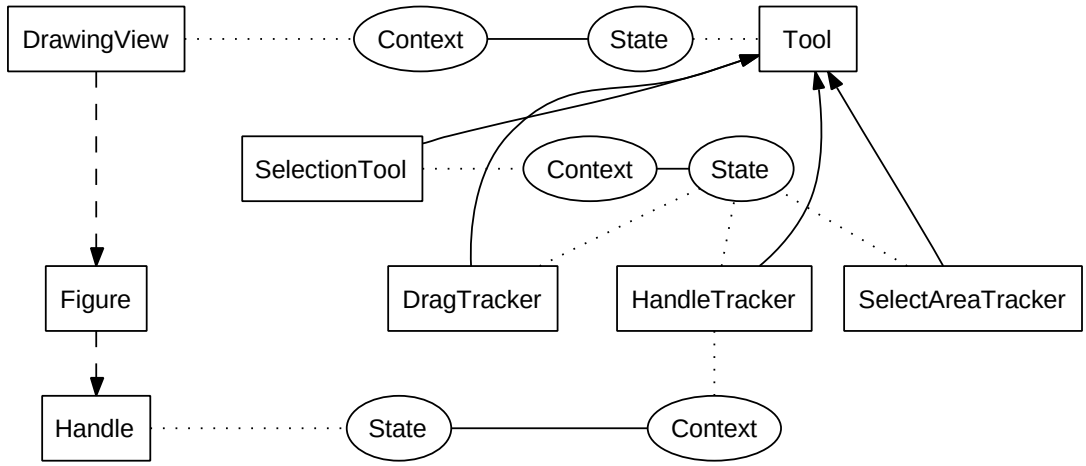
`Handle` est un adapter permettant à `Figure` de répondre aux événements souris (figure 4.3), ce qui rend possible graphiquement la manipulation directe de propriétés comme la taille d'une figure. `Handle` constitue de fait un outil particulier à `Figure`.

4.2.3.6 Factory Method

Le motif FACTORY METHOD déporte la création d'un objet dans une méthode normale, donc sujette à redéfinition. C'est un moyen de définir un constructeur polymorphique.

Ce motif permet à chaque `Figure` d'instancier à la demande ses propres `Handle`s, spécifiques à ses propriétés (figure 4.3).

Illustration 4.4 – Implication des motifs autour de `Tool`



4.2.3.7 State

Le motif STATE permet à un objet de changer de comportement dynamiquement suivant un état interne, en délégant à celui-ci les requêtes.

Une occurrence fondamentale est la relation entre `DrawingView` et `Tool` (figure 4.4). Ici `DrawingView` transmet les événements de l'interface graphique à l'outil courant. Celui-ci décide de l'action appropriée suivant le contexte de la vue (par exemple, la sélection courante). Il faut noter que cette implémentation s'appuie sur le médiateur ci-dessus : en particulier celui-ci a la responsabilité du changement d'état (outil).

Une occurrence particulièrement intéressante est fournie par `SelectionTool`, l'outil par défaut permettant de manipuler les figures. Suivant le contexte de la vue (particulièrement sous le curseur), `SelectionTool` choisit son comportement entre trois états – `DragTracker` (sélection d'une figure et déplacement), `SelectAreaTracker` (sélection d'une zone) et `HandleTracker` (sélection d'un `Handle`) – puis délègue tous les événements à l'outil choisi jusqu'à fin de l'activité.

La relation de `HandleTracker` à `Handle` constitue une troisième occurrence. En effet après récupération et sélection d'un `Handle` auprès de la figure visée (grâce au motif Factory Method), `HandleTracker` se charge à son tour de transmettre les événements au `Handle`.

4.2.3.8 Template Method

Le motif Template Method décrit l'usage de l'héritage et de la redéfinition pour définir des algorithmes génériques et les spécialiser dans les sous-classes. Par son principe, c'est donc un motif d'un usage standard et très fréquent dans les applications à objets et en particulier dans les cadriciels. Nous ignorons ce motif par la suite, le considérant comme un cliché des langages à classes (section 2.2.1.3 page 16).

4.2.3.9 Synthèse

Le tableau 4.1 fait la synthèse de l'implication des différents motifs dans les entités du noyau. Cette représentation ne préjuge cependant pas de la complexité de l'implémentation des motifs dans chaque entité : l'impact d'un motif sur une entité dépend du rôle joué par l'entité. Par exemple l'entité `Figure` est impliquée dans de nombreux motifs dont Adapter : mais ce dernier n'a pas d'impact sur `Figure` puisqu'il a pour objectif d'adapter sans modifier la classe.

Tableau 4.1 – Implication de différents motifs dans les entités du noyau JHotDraw

Entités→ Motifs ↓	DrawingEditor	DrawingView	Drawing	Figure	Tool	Handle
Mediator	*	*			*	
Observer		*	*	*		
Strategy		*				
Prototype				*	*	
Adapter				*		*
Factory Method				*		*
State		*		*	*	*
Densité des rôles[a]	1	4	1	5	3	3

[a] Voir section 4.2.4.2

Dans le noyau, deux catégories de motifs se distinguent par leur finalité. Les motifs Mediator et Observer fixent des protocoles de collaborations génériques. Les motifs Strategy, Prototype, State, Factory Method et Adapter ciblent l'extension du cadriciel en découplant certaines fonctionnalités. En particulier les quatre derniers facilitent la définition de nouvelles `Figure` et de leurs `Handle` spécifiques.

Le détail de la collaboration de `Handle` avec `Figure` et `Tool` via les motifs State, Adapter et Factory Method est à ce sujet intéressant : State permet la sélection dynamique et l'exécution d'un outil `Handle`, lequel est un Adapter traduisant les événements en actions spécifiques à la figure, qui a elle-même fourni ce `Handle` grâce à Factory Method. Cette description montre que les motifs permettent de décrire une collaboration de façon synthétique.

Le programmeur peut utiliser l'information des patrons et motifs pour comprendre le fonctionnement du noyau et les principes impliqués dans l'extension. Cependant seule une partie des collaborations sont décrites par des motifs.

4.2.4 Mesures et interprétation de la densité des motifs dans le noyau

Pour clore cette analyse du noyau et de ses motifs, nous présentons quelques réflexions sur la mesure de la densité et son utilité en tant que métrique. La densité est un rapport entre deux quantités : une mesure liée aux motifs et une mesure liée au code source. Une métrique de densité ainsi que son interprétation dépend donc de ces deux mesures. Chacune de ces mesures dépend de deux paramètres.

- Unité de la mesure – qu'est-ce qui est compté ?
- Portée de la mesure – dans quel espace compter ?

Pour que la métrique de densité ait un sens, il faut que l'espace de comptage des motifs soit égal ou inclus dans l'espace du code source. Nous mesurons la densité des motifs dans le noyau : seuls les motifs impliqués dans le noyau sont évidemment comptés. Les métriques que nous présentons jouent sur ces paramètres.

4.2.4.1 Densité des motifs

Une version simple de la métrique de densité consiste à compter les *occurrences** des motifs et les éléments (classes, interfaces) de la conception du programme. Cependant plusieurs métriques peuvent être définies suivant la portée de la mesure.

- Densité relative : espace défini comme un sous-ensemble arbitraire d'éléments.
- Densité absolue : espace défini comme l'ensemble des éléments.

Une mesure de densité dans le noyau de JHotDraw correspond donc à une densité *relative*, puisque nous avons sélectionné un ensemble d'*entités** , qui se subtitue aux classes et interfaces de l'implémentation et ne prend pas en compte les classes utilitaires (ce qui impliquerait d'autres motifs). Suivant la section 4.2.2, nous comptons six entités (interfaces) dans le noyau. Suivant la section 4.2.3, nous comptons neuf occurrences de motifs (dont deux occurrences de Observer et deux de Strategy). La densité des motifs dans le noyau de JHotDraw est donc de 1,5 – soit trois motifs pour deux entités.

À titre de comparaison, l'étude de [Gamma et Beck, 2002] présente six motifs pour un sous-ensemble de cinq classes du cadriciel JUnit. Cette étude présente donc aussi une densité supérieure à 1 (il est possible que d'autres motifs soient impliqués mais non présentés).

> Définition 4.3 – Forte densité de motifs de conception
>
> Nous définissons comme seuil de forte densité une densité de motifs supérieure à 1. Dans un tel ensemble chaque élément est impliqué en moyenne dans au moins un motif. L'attente est donc que la majorité des relations et des extensions sont gérées par des motifs.

Cependant cette métrique ne peut pas être prise, à notre avis, comme une mesure de complexité du code ou de la conception. En particulier elle ne tient pas compte de la difficulté d'implémentation d'un motif et de son impact sur les classes.

4.2.4.2 Densité des rôles

La notion de densité peut être raffinée en comptant les *rôles* des motifs : cette mesure rend mieux compte de l'implication des motifs dans les classes et interfaces.

La mesure la plus simple consiste simplement à compter, pour une entité, le nombre de rôles dans lesquelles celle-ci est impliquée. Ceci correspond à faire le sous-total pour chaque colonne du tableau 4.1.

Score d'implication. Cette mesure peut être interprétée comme le score d'implication d'une entité dans les motifs du noyau. L'entité `Figure` obtient le score de 6 devant l'interface `DrawingView` (5). Ensuite viennent `Tool` et `Handle`, puis `DrawingEditor` et `Drawing`.

Ce score établit donc un classement entre les entités du noyau : les entités au plus haut score sont les plus impliquées dans des relations avec d'autres entités. On peut donc s'attendre à ce que ces entités soient plus complexes. Cependant, comme pour la densité des motifs, cette métrique n'est pas une mesure de la complexité des motifs et des implémentations mais un simple indicateur.

À la manière de [Gamma et Beck, 2002], nous remarquons que l'abstraction centrale au domaine (`Figure`) obtient le score le plus élevé. Ceci suggère un usage de la mesure de densité des rôles comme moyen de détection des entités d'une application : seules les classes et interfaces obtenant un score non nul (ou supérieur à un seuil) peuvent être considérées comme des entités et faire partie du noyau.

4.2.4.3 Synthèse

Ces définitions pour des métriques de densité constituent, à notre avis, de bons indicateurs pour juger de la maturité du code et distinguer les entités essentielles d'un programme. Elles pointent vers les zones potentielles de complexité dans les relations entre les entités mais ne mesurent pas cette complexité.

4.2.5 Synthèse de l'étude par les motifs

Nous avons présenté une démarche documentant le noyau de JHotDraw par ses entités ainsi que son fonctionnement et son extension par des motifs de conception. La forte densité des motifs dans le noyau permet de documenter et comprendre en partie le cadriciel de façon synthétique. Les métriques de densité constituent des indicateurs pour guider le programmeur dans sa tâche de conception.

4.3 Impact de la densité des motifs sur l'implémentation objet

Le noyau de JHotDraw présente une forte densité de motifs suivant la définition (section 4.2.4.1) : chaque entité est impliquée dans au moins un motif. Cependant l'implémentation des motifs a un impact différent sur chaque entité. Suivant les principes d'encapsulation et d'économie de notre noyau (section 4.2.1), cet impact est considéré comme minime et ignoré tant qu'il est complètement interne au noyau. À l'inverse nous avons posé comme principe la modularité des extensions non intégrées au noyau.

Dans cette section nous abordons donc les conséquences de l'implémentation, si possible modulaire, de motifs dans ce contexte de forte densité. Notre étude repose sur l'ajout de préoccupations supportées par ces nouveaux motifs. Ces préoccupations font parties du cadriciel original mais sont exclues du noyau car jugées accessoires. Les réintégrer permet de nous comparer au cadriciel original pour en comprendre les choix et les modifications. Cette démarche souligne aussi comment la traçabilité des choix de conception est perdue dans l'implémentation.

Cette étude est réalisée dans le cadre purement objet. Nous commençons par détailler la fonction et l'implémentation d'une préoccupation du noyau, l'invalidation. Nous présentons ensuite des extensions et analysons les conséquences de leurs implémentations sur le noyau, en particulier sur cette préoccupation et ses motifs.

4.3.1 Détail d'une préoccupation du noyau : l'invalidation

La préoccupation d'invalidation est intégrée au rafraîchissement de la vue : elle est précisément supportée par les occurrences du motif Observer présentées en section 4.2.3.2. L'explication par les patrons donne une abstraction de l'invalidation (la notification) mais n'en expose pas les spécifications.

4.3.1.1 Fonction de l'invalidation

L'invalidation est une fonction liée au rafraîchissement d'une vue, c'est-à-dire à la mise à jour de la vue quand une de ses figures change son apparence ou sa position. Cependant la mise à jour suit un processus particulier dans les applications graphiques : d'une part celle-ci réagit de façon asynchrone à une notification de changement, de sorte qu'elle répond à plusieurs notifications à la fois – ceci est nécessaire pour éviter la surcharge des notifications face à la durée relativement longue du processus de dessin (cette contrainte est imposée par les cadriciels graphiques tel Swing) ; d'autre part le cadriciel graphique demande si possible de miniminer la zone à redessiner, afin d'accélérer le processus et d'éviter des flashs de la vue – c'est une optimisation classique appelée clipping (découpage).

Le processus de rafraîchissement se décompose alors en deux étapes : premièrement capturer les changements des figures pour calculer la zone de découpage et la transmettre à Swing ; deuxièmement dessiner dans le contexte graphique à la requête de Swing. Nous appelons *invalidation* la première étape. Son rôle est donc de collecter les zones invalides suite aux modifications des figures (création, déplacement, changement d'apparence), de calculer la zone de découpage totale et d'envoyer celle-ci avec la requête `repaint` à Swing.

4.3.1.2 Conception initiale et implémentation

Dans JHotDraw, les figures notifient la zone invalidée par leurs modifications. Elles définissent cette zone comme étant leur rectangle englobant. Le calcul de la zone de découpage se fait alors par union des zones invalidées entre deux requêtes de mises à jour. Cette zone est mémorisée dans la vue. Enfin les requêtes de mise à jour sont envoyées par la vue (via la relation à `JPanel`) : elles marquent la fin de l'exécution d'un outil.

L'implémentation repose en particulier sur l'enchaînement de deux occurrences du motif Observer. Cet enchaînement respecte le découplage imposé par le dessin entre la vue et les figures. La première occurrence lie les `Figure`s à leur `Drawing` propriétaire :

- quand une figure est ajoutée à un dessin (après sa création par exemple), le dessin est enregistré comme observateur de la figure (et inversement en cas d'effacement) ;
- lors d'une modification, la figure notifie le dessin avec la zone invalidée (rectangle englobant) ;
- sur notification d'une figure, le dessin transmet la zone invalidée à sa vue suivant le processus ci-dessous.

La seconde occurrence prend donc en charge la transmission de `Drawing` à `DrawingView` :

- un dessin enregistre la vue comme observateur ;
- sur notification du dessin, la vue ajoute le rectangle englobant à sa zone de découpage.

Les deux occurrences de Observer implémentent donc l'étape de récupération et de calcul de la zone de découpage dans le processus d'invalidation.

La deuxième étape de l'invalidation concerne surtout `DrawingView`. La requête de mise à jour Swing est effectuée par `DrawingView`, suite à quoi la zone de découpage est remize à zéro. Cette requête peut être initiée soit par `DrawingView`, soit par l'outil courant : la notification de `Tool` à `DrawingView` est possible grâce au motif Mediator de `DrawingEditor` (cette notification indique en fait une occurrence cachée de Observer).

4.3.2 Variation intra-motif

La variation intra-motif désigne un changement dans l'implémentation du motif en réponse à de nouvelles contraintes, sans pour autant changer la fonction supportée par le motif. Il peut s'agir en particulier de variations standards, telles qu'elles sont décrites dans les sections *Implémentation* des patrons du *GoF*.

Nous nous intéressons pour notre étude au cas où la variation est liée à l'utilisation d'un autre motif dans l'implémentation (suivant la classification des relations de Zimmer, section 2.2.3.2 page 22). Ce cas montre l'opportunité et le risque associés à l'utilisation du motif Singleton.

4.3.2.1 Scénario

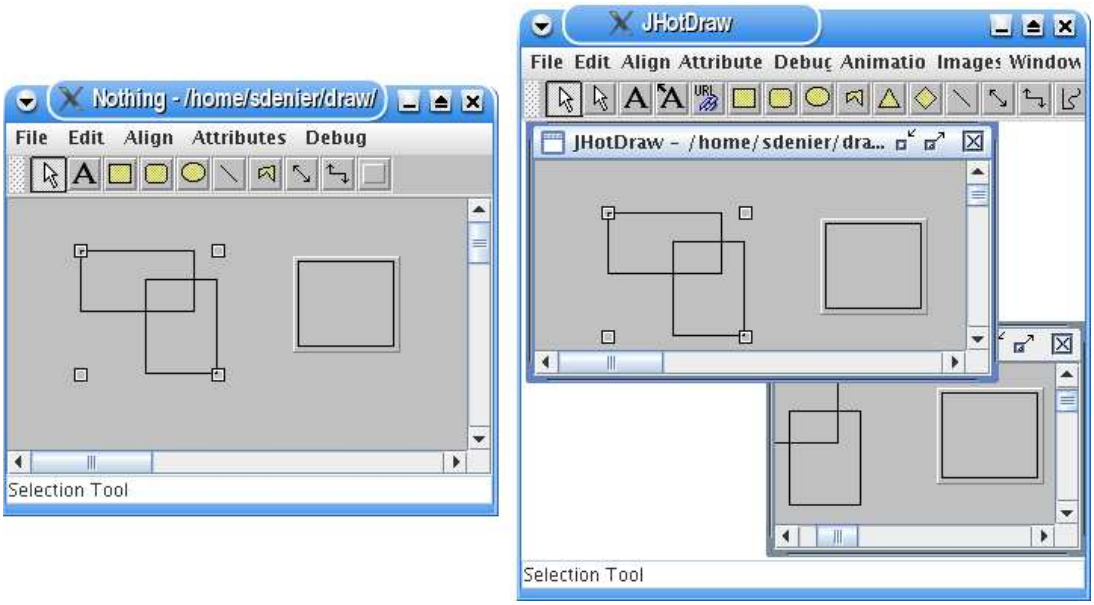
Les applications construites directement à partir du noyau de JHotDraw n'affichent qu'une seule fenêtre, ce qui implique qu'à tout moment une seule vue est suffisante pour afficher un dessin. Dans le noyau, l'entité `DrawingView` est donc une occurrence de Singleton.

Une extension de JHotDraw propose de construire des applications *MDI* (*Multiple Document Interface*), permettant d'afficher plusieurs vues dans l'application (figure 4.5). Les spécifications indiquent que ceci permet 1) d'afficher plusieurs dessins de façon simultanée, mais aussi 2) d'avoir plusieurs vues sur le même dessin. Une conséquence évidente et immédiate est que `DrawingView` perd sa nature de Singleton.

4.3.2.2 Impact

L'usage du motif Singleton a pour conséquence la définition d'un point d'accès universel à l'instance de la vue. Dans le cadre de l'invalidation et en particulier de la seconde occurrence Observer entre `Drawing` et `DrawingView` (section 4.3.1.2), ceci rend l'implémentation très simple (exemple 4.1). Le sujet `StandardDrawing` notifie directement la vue (ligne 4). Il n'y a pas besoin d'implémenter une liste d'observateurs et les méthodes associées à l'enregistrement.

Illustration 4.5 – Deux applications issues de JHotDraw : à gauche une version issue directement du noyau avec une seule vue par dessin ; à droite une version *MDI* avec plusieurs vues sur le même dessin.



L'extension *MDI* de JHotDraw introduit un nouveau niveau d'indirection entre les entités `DrawingEditor` et `DrawingView`, basé sur la classe `JInternalFrame` de Swing. `DrawingEditor`, qui est le composant graphique principal, délègue alors une partie de l'affichage aux `JInternalFrame`, qui à leur tour délèguent le dessin à leur propre `DrawingView`. La gestion des `JInternalFrame` est prise en charge de façon modulaire par une sous-classe de `DrawApplication`. La relation de MEDIATOR jouée par `DrawingEditor` n'est pas perturbée.

Si l'extension principale a donc un impact modulaire, il n'en est pas de même si on considère l'occurrence de OBSERVER entre `Drawing` et `DrawingView`. Du fait du retrait du motif SINGLETON, l'accès universel n'est plus possible. De plus, il peut désormais y avoir plusieurs vues par dessin. Ces deux contraintes modifient profondément l'implémentation du sujet `Drawing` (exemple 4.2). Il faut mémoriser la relation aux observateurs (lignes 3–4), introduire les méthodes d'enregistrement (lignes 7–8) ainsi que leur appel (non montré) et changer la notification simple en diffusion de la notification (ligne 10).

Bien que les changements de l'occurrence OBSERVER soient limités à la seule classe `StandardDrawing`, ceux-ci sont la conséquence du retrait du motif SINGLETON dans la classe `StandardDrawingView`. Le changement initial a donc eu un impact indirect et invasif sur `StandardDrawing`, lié au mélange de l'implémentation du motif avec la classe. Ces changements ne sont pas modulaires.

On peut critiquer la naïveté de la solution SINGLETON du noyau original. En effet le changement ci-dessus, qui correspond à la généralisation du motif OBSERVER, est compatible avec le cas SINGLETON. Cette solution pourrait donc être intégrée directement au noyau. Mais cela ne correspond pas à notre principe d'économie (section 4.2.1) : rajouter cette complexité est inutile pour rendre le noyau fonctionnel.

Exemple 4.1 – Motif OBSERVER : implémentation simple du sujet `Drawing` avec l'observateur SINGLETON `DrawingView`

```
1  class StandardDrawing (...) implements Drawing {
2      (...)
3      public void figureInvalidated(FigureChangeEvent e){
4          StandardDrawingView.instance().drawingInvalidated(
5                  new DrawingChangeEvent(this,
6                                         e.getInvalidatedRectangle()));}
7  }
```

Exemple 4.2 – Motif OBSERVER : implémentation du sujet `Drawing` avec plusieurs observateurs `DrawingView`

```
1  class StandardDrawing (...) implements Drawing {
2      (...)
3      private Vector<DrawingView> observers
4          = new Vector<DrawingView>();
5      // quand une vue est liee au dessin, elle doit appeler
6      // cette methode pour etre enregistree comme observateur
7      public void addObserver(DrawingView view) { ... }
8      public void removeObserver(DrawingView view) { ... }
9      public void figureInvalidated(FigureChangeEvent e){
10         for( DrawingView view: observers )
11             view.drawingInvalidated(
12               new DrawingChangeEvent(this,
13                                      e.getInvalidatedRectangle()));}
14 }
```

4.3.2.3 Synthèse

L'exemple des motifs OBSERVER et SINGLETON montre l'opportunité et le risque liés à l'utilisation d'un motif par un autre. SINGLETON permet de simplifier l'implémentation en créant un accès universel. L'exploitation de cette propriété expose cependant au risque, pour les autres motifs, de variations internes. Or ces variations ne sont pas modulaires car l'implémentation des motifs est dispersée et mélangée avec les classes.

4.3.3 Transformation de motif

La transformation de motif est un changement visible dans la façon dont le motif fonctionne et est utilisé. L'implémentation est donc modifiée mais aussi l'interface. De nouvelles classes peuvent être impactées. Il faut adapter le code faisant usage de ce motif.

Pour notre étude sur la densité, nous nous intéressons au cas où deux motifs mis en présence interagissent. Leur interaction aboutit à un changement de fonctionnement de l'un des motifs : il est transformé en un autre motif.

4.3.3.1 Scénario

Nous examinons le cas de deux extensions génériques de l'entité `Figure` : il s'agit des classes abstraites `CompositeFigure` et `DecoratorFigure`, basées respectivement sur les motifs Composite et Decorator. Ces extensions permettent de construire des arbres de `Figure`s (figure 4.6), contrairement au noyau original où toutes les `Figure`s étaient directement sous l'entité `Drawing`.

Les classes `CompositeFigure` et `DecoratorFigure` définissent l'infrastructure abstraite des motifs et offrent un comportement par défaut neutre. Il est nécessaire de les spécialiser au cas par cas. Ainsi nous introduisons deux extensions concrètes de ces classes, respectivement `GroupFigure` et `BorderDecorator`. `GroupFigure` permet de manipuler uniformément un ensemble de figures. `BorderDecorator` améliore simplement l'apparence d'une figure par une bordure (figure 4.5, figure 4.6, coin supérieur gauche).

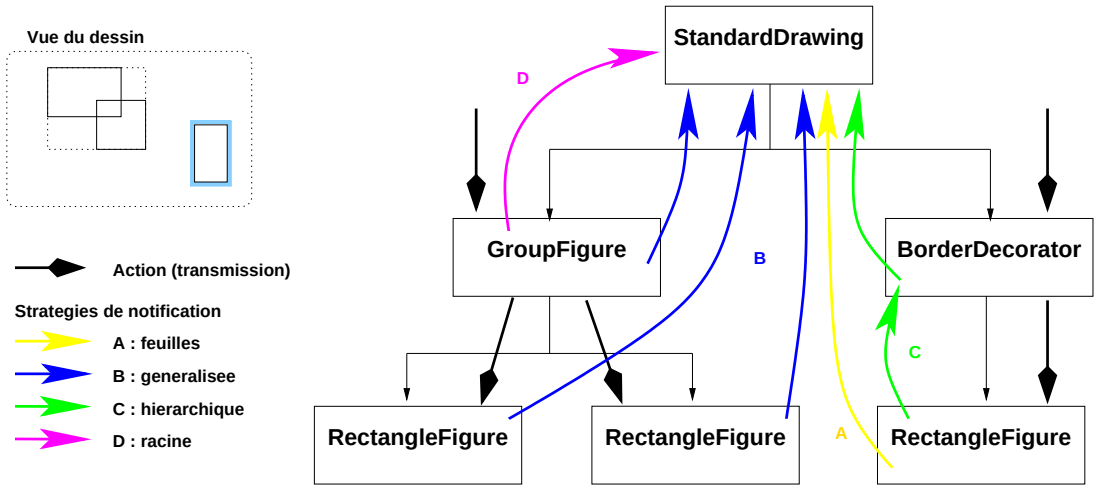
4.3.3.2 Effet sur l'invalidation : stratégies de notification

Le changement d'apparence induit par la classe `BorderDecorator` a un impact sur la zone d'invalidation de la figure décorée. En effet toute modification qui implique de redessiner la figure implique aussi de redessiner le décorateur autour. À chaque invalidation il faut donc prendre en compte la zone délimitée par la bordure, qui englobe la zone de la figure seule.

Ceci pose problème car il y a disjonction entre deux paramètres de la notification d'invalidation : seul `BorderDecorator` est capable de définir la zone invalidée, connaissant la taille de sa bordure et la zone invalidée de la figure décorée ; mais seule la figure décorée peut juger sans ambiguïté qu'un changement est invalidant et lancer la notification.

Par exemple des actions génériques telles que modifier la couleur de fond ou le style de flèche n'ont pas d'effet sur les figures de contour ou sans flèche. Il n'est pas nécessaire d'invalider les figures dans ce cas. Par contre une action aussi universelle que déplacer une figure invalide à coup sûr celle-ci.

Illustration 4.6 – Arbre de `Figure`s utilisant les motifs Composite et Decorator. L'affichage de l'arbre est montré dans le coin haut-gauche. Ce diagramme objet montre différentes stratégies de notification d'invalidation (section 4.3.3.2)



Considérant ces cas, la figure 4.6 expose quatre stratégies pour la notification d'invalidation :

A. la notification par les feuilles ignore les composites et décorateurs dans la hiérarchie, ce qui a pour risque de fausser les zones d'invalidation et l'affichage (dans le cas de `BorderDecorator`) ;

B. la notification généralisée notifie toute action, y compris dans les composites et décorateurs, ce qui implique des notifications redondantes voire inutiles dans certains cas (comme pour les attributs de flèche) ;

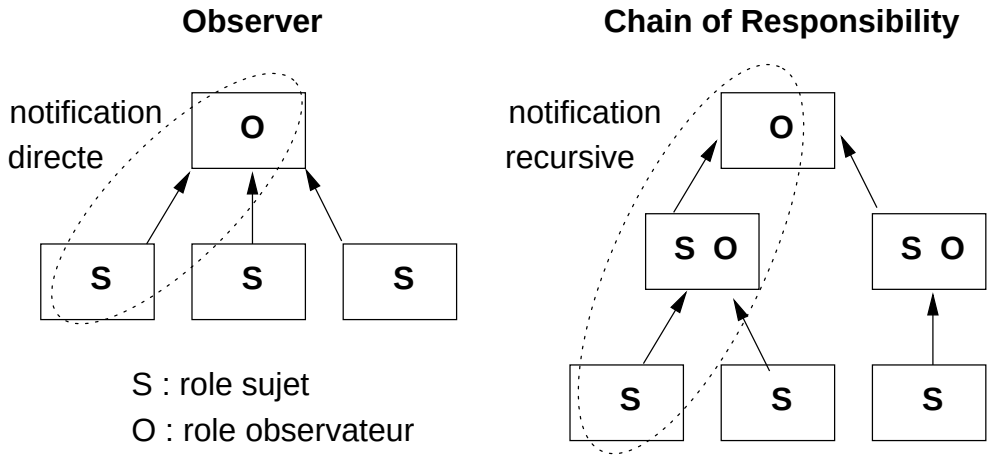
C. la notification hiérarchique indique qu'une `Figure`, au lieu de notifier directement `Drawing`, notifie son parent direct dans l'arbre – le parent peut modifier si besoin et transmettre à son tour la notification, qui remonte récursivement jusqu'à `Drawing` ;

D. la notification à la racine (d'appel) est spécifique à certains cas, tel le déplacement de figures, pour lesquelles il y a sans ambiguïté modification et où un processus de notification récursif n'est pas nécessaire.

En l'absence des motifs Composite et Decorator, il n'y a que des feuilles sous `Drawing`. La stratégie des feuilles (A) est donc suffisante (et économe) pour le noyau.

La stratégie hiérarchique (C) est conceptuellement la plus intéressante puisqu'elle est à la fois valide dans tous les cas (contrairement aux stratégies A et D) et plus précise que la stratégie généralisée (B). C'est cette stratégie qui est implémentée dans le cadriciel original pour prendre en compte les motifs Composite et Decorator et le cas `BorderDecorator`.

Illustration 4.7 – Transformation du motif Observer en motif Chain of Responsibility. La notification directe du sujet à l'observateur est remplacée par une notification récursive remontant la hiérarchie des sujets–observateurs



4.3.3.3 Impact sur la conception et l'implémentation

Pour l'invalidation dans le noyau, la première occurrence de Observer entre `Figure` et `Drawing` supporte la stratégie des feuilles. La relation est directe `Figure` et `Drawing`. Avec les motifs Composite et Decorator, cette relation doit être transformée en relation de parentalité

remontant l'arbre des `Figure`s. Chaque figure parente observe donc ses propres sous-figures : en recevant une notification, elle peut la modifier avant de la transmettre à son tour à son parent. `Drawing` reste l'observateur final. Les figures parentes sont donc à la fois sujet et observateur dans une suite récursive d'Observer. Sur l'ensemble des classes impliquées, cette conception correspond au motif Chain of Responsibility (figure 4.7).

L'interaction des motifs Composite, Decorator avec Chain of Responsibility crée conceptuellement deux récursions, imbriquées et en sens inverse l'une de l'autre. La première récursion est celle de l'action qui descend à partir des figures de premier niveau (sous `Drawing`) jusqu'aux feuilles. Durant cette récursion, chaque notification lancée remonte récursivement la chaîne jusqu'à `Drawing`. Une action peut donc déclencher plusieurs notifications récursives.

L'implémentation du sujet de la relation hiérarchique n'est pas fondamentalement différente de la version du noyau (exemple 4.3). Chaque `Figure` dispose déjà d'un champ permettant d'enregistrer la référence à l'observateur (ligne 2), représentée par l'interface `FigureChangeListener`. Il faut cependant changer l'enregistrement de l'observateur, qui n'est plus forcément `Drawing`.

Exemple 4.3 – Motif Observer : implémentation du sujet `Figure`. `AbstractFigure` définit l'infrastructure générale. À chaque action invalidante (ligne 4) correspond une ou plusieurs notifications (lignes 5–7). L'observateur est abstrait par l'interface `FigureChangeListener`. `figureInvalidated(...)` (ligne 12) est la méthode de notification pour l'invalidation

```
1  abstract class AbstractFigure implements Figure { // sujet
2      private FigureChangeListener observer;
3      (...)
4      public void moveBy(int dx, int dy){ invalidate(); (...) }
5      public void invalidate(){
6        Rectangle r = displayBox(); // zone invalidee
7        observer.figureInvalidated(new FigureChangeEvent(this, r));
8      }
9      public abstract Rectangle displayBox();
10 }
11 interface FigureChangeListener { // observateur
12     public void figureInvalidated(FigureChangeEvent e); }
```

L'impact est plus sensible sur les classes `CompositeFigure` et `DecoratorFigure`. Celles-ci doivent en effet implémenter leur rôle d'observateur (exemple 4.4). `FigureChangeListener` regroupe donc les classes `Drawing`, `CompositeFigure` et `DecoratorFigure`. `CompositeFigure` et `DecoratorFigure` implémentent par défaut la transmission des notifications (`CompositeFigure`, lignes 4–5). La modification d'une notification est faite par simple redéfinition dans le cas de BorderDecorator (lignes 13–16).

4.3.3.4 Synthèse

La transformation du motif Observer en Chain of Responsibility en présence des motifs Composite et Decorator est complexe. L'impact des motifs est renforcé par leur interaction.

Exemple 4.4 – Motif Chain of Responsibility : implémentation de la transmission dans les classes composites et décoratrices. `CompositeFigure` et `DecoratorFigure` héritent toutes les deux de `AbstractFigure` pour être sujets et implémentent `FigureChangeListener` pour être observateurs. Le comportement par défaut de `figureInvalidated(...)` est la transmission de la notification (`CompositeFigure`, lignes 4–5). `BorderDecorator` doit redéfinir cette méthode pour modifier la zone invalidée (lignes 13–16)

```
1  class CompositeFigure extends AbstractFigure // Composite
2    implements FigureChangeListener {
3       (...)
4       public void figureInvalidated(FigureChangeEvent e){
5           observer.figureInvalidated(e);} // transmission
6  }
7  class GroupFigure extends CompositeFigure {...}// extension
8
9  class DecoratorFigure extends AbstractFigure // Decorator
10   implements FigureChangeListener {...}
11 class BorderDecorator extends DecoratorFigure {// extension
12      (...)
13      public void figureInvalidated(FigureChangeEvent e){
14          Rectangle r = e.getInvalidatedRectangle();
15          r.grow(factorx, factory); // ajoute la taille de la bordure
16          observer.figureInvalidated(new FigureChangeEvent(this, r));}
17 }
```

Du point de vue conception, l'impact de la spécification de `BorderDecorator` se généralise à l'ensemble des classes : c'est un exemple où la traçabilité entre la spécification et le choix de conception est difficile. Cependant la solution est élégante et facile à utiliser, simplement par héritage et redéfinition.

Du point de vue implémentation, la stratégie de notification doit être implémentée à l'échelle de la hiérarchie des classes et à l'exclusion des autres stratégies. Elle est donc imposée à toutes les notifications. Le grain de décomposition imposé par les classes et l'héritage ne permet pas de raffiner et de choisir au cas par cas la stratégie à utiliser.

4.3.4 Réutilisation de motif

L'impossibilité d'avoir des motifs génériques réutilisables (section 2.2.2.3 page 19) n'interdit pas la réutilisation d'une occurrence locale. Le coût d'implémentation d'un motif incite à réutiliser les éléments en place. Plus la densité est grande, plus les opportunités sont possibles. Cependant, l'exploitation de ces opportunités implique des compromis qui violent la séparation des préoccupations et fragilisent le code.

4.3.4.1 Fusion de préoccupations

La fusion de préoccupations indique qu'une seule occurrence d'un motif supporte plusieurs préoccupations. Mais ceci demande de mélanger les préoccupations pour qu'elles cohabitent dans

l'implémentation du motif. La réutilisation de code est privilégiée au détriment de la séparation des préoccupations. Ce résultat est le fruit d'un compromis conjoncturel.

Scénario. Notre exemple concerne une extension de JHotDraw permettant de connecter des figures avec d'autres figures. Une figure connectée réagit aux modifications d'une autre figure. Par exemple, il est possible d'attacher du texte à une figure : le texte suit la figure dans ses déplacements. Cette fonction est aussi utile aux lignes de connexion : celles-ci connectent deux figures par une polyligne et recalculent leur placement en cas de changement des extrémités.

Conception. Le motif OBSERVER permet de supporter ces dépendances entre figure et figure connectée. Cependant la spécification est différente par rapport à l'occurrence de OBSERVER pour l'invalidation (section 4.3.1.2). La dépendance de connection est transversale – de figure à figure – et non plus hiérarchique – de la figure au dessin.

L'extension modulaire des classes implémentant `Figure` avec une nouvelle occurrence de OBSERVER est possible par héritage ou grâce au motif DECORATOR. Cependant une autre solution est implémentée dans le cadriciel original JHotDraw. La seule occurrence de OBSERVER est définie pour gérer les deux préoccupations. Celles-ci partagent donc une partie de l'implémentation – en particulier chaque sujet a un unique ensemble d'observateurs pour les deux fonctions – mais voient leurs préoccupations fusionner. Chaque observateur d'une des deux préoccupations doit alors traiter les notifications de l'autre préoccupation.

Impact. La fusion est visible dans l'interface des observateurs `FigureChangeListener` (exemple 4.5). Ceci se traduit par une invasion dans les observateurs : les figures de connexion (telles que le texte, les lignes de connexion) doivent traiter la notification d'invalidation ; réciproquement le dessin et les figures hiérarchiques doivent traiter les notifications liées aux connections. En pratique, les classes observateurs « ignorent » les notifications qui ne les concernent pas par des implémentations vides. Cet impact minime montre cependant que les deux préoccupations sont mélangées entre elles et avec les `Figure`s.

Exemple 4.5 – Motif OBSERVER : interface `FigureChangeListener` du rôle observateur fusionnant les préoccupations d'invalidation et de connexion

```
1  interface FigureChangeListener {
2      // preoccupation invalidation
3      public void figureInvalidated(FigureChangeEvent e);
4      // preoccupation connexion
5      public void figureChanged(FigureChangeEvent e);
6      public void figureRemoved(FigureChangeEvent e);
7      public void figureRequestRemove(FigureChangeEvent e);
8      public void figureRequestUpdate(FigureChangeEvent e); }
```

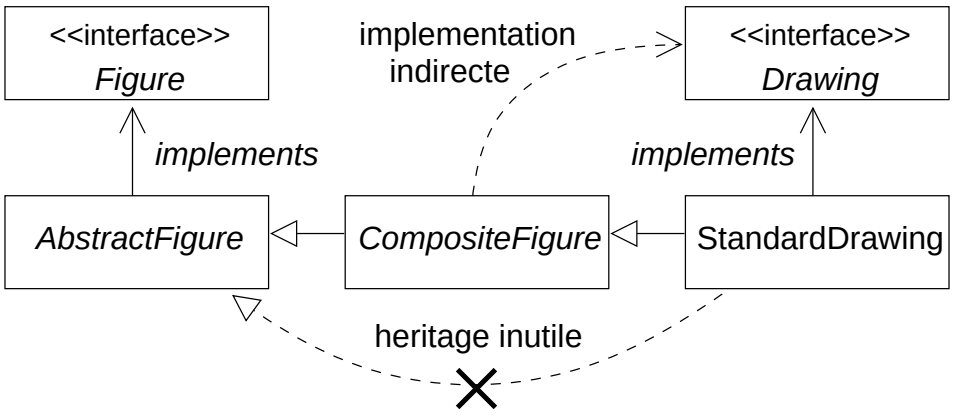
La fusion simplifie l'implémentation en réutilisant le code encombrant du motif (la gestion des observateurs par le sujet). Mais elle demande une compatibilité des préoccupations et les mélange : l'impact est combinatoire car il implique toutes les classes des préoccupations visées.

4.3.4.2 Réutilisation avec l'héritage simple

Le mécanisme d'héritage joue trois rôles complémentaires dans les langages à classe : il permet la factorisation et la réutilisation de code, il définit une relation de sous-classe, enfin il supporte le polymorphisme. Un usage normal de l'héritage prend en compte ces trois rôles. Un usage détourné privilégie un des rôles sans considérer les autres. En particulier dans les langages où l'héritage simple est le seul mécanisme de réutilisation, ce rôle a une importance prédominante.

La classe `StandardDrawing`, qui implémente l'interface `Drawing`, illustre ce point dans le cadriciel original JHotDraw. L'entité `Drawing` est l'intermédiaire unique entre `DrawingView` et l'ensemble des `Figure`s : elle contient et manipule donc des `Figure`s. Cette vocation la rapproche du motif COMPOSITE et donc de la classe `CompositeFigure`. De fait, dans le cadriciel original, `StandardDrawing` hérite de `CompositeFigure` afin de bénéficier de l'implémentation existante. Pour démontrer le biais de cette relation d'héritage, nous évaluons deux intuitions : l'implémentation indirecte de `Drawing` par `CompositeFigure` et l'héritage inutile de `AbstractFigure` par `StandardDrawing` (figure 4.8).

Illustration 4.8 – Biais de la réutilisation du motif COMPOSITE par héritage dans JHotDraw. `CompositeFigure` est une implémentation indirecte de `Drawing`. `StandardDrawing` hérite inutilement de `AbstractFigure`



La première intuition est la fonction d'implémentation indirecte jouée par `CompositeFigure` vis-à-vis de `Drawing`. Bien que ces deux modules ne soient pas directement en relation, 82% des 39 méthodes de `Drawing` ont leur équivalent défini dans `CompositeFigure` (ce qui représente aussi 78% des méthodes de `CompositeFigure`). Cette relation est confirmée par le fait que `StandardDrawing` n'implémente que 11 des méthodes de `Drawing` : en héritant de `CompositeFigure`, `StandardDrawing` implémente automatiquement la plupart des méthodes de `Drawing`. En d'autres termes, l'interface `Drawing` copie l'interface de `CompositeFigure`, ce qui crée une relation implicite entre les deux. Ceci rend la hiérarchie de `StandardDrawing` plus fragile puisque, en cas de modification dans `CompositeFigure`, `Drawing` peut être affectée et réciproquement.

La seconde intuition est l'inutilité de l'héritage par `StandardDrawing` de `Figure` et `AbstractFigure`, via `CompositeFigure`. `StandardDrawing` hérite par cette relation des méthodes et attributs

d'une figure. Cette confusion entre `Figure` et `Drawing` est particulièrement visible dans l'implémentation de l'invalidation : dans le premier OBSERVER de l'invalidation, `Figure` implémente le rôle sujet dont `StandardDrawing` est l'observateur. Ce rôle sujet hérité par `StandardDrawing` est donc inutile. Par contre `StandardDrawing` implémente bien un rôle sujet dans le second OBSERVER de l'invalidation. Cette confusion des rôles dans la classe `StandardDrawing` n'a en pratique pas d'effet sur le comportement de ses instances : celles-ci ne sont jamais référencées en tant que `Figure` et n'utilisent pas les méthodes héritées de `Figure`. En particulier le rôle sujet de `Figure` n'est pas sollicité dans l'usage de `StandardDrawing`.

Pour ces raisons nous concluons que l'héritage de `CompositeFigure` par `StandardDrawing` privilégie la réutilisation du motif COMPOSITE au détriment du sous-classage.

4.3.4.3 Synthèse sur la densité et la réutilisation

En l'absence d'implémentation modulaire des motifs, une forte densité peut offrir des opportunités de réutilisation des occurrences locales. Ceci implique cependant une proximité des préoccupations supportées par les motifs : nous avons vu comme thèmes fédérateurs l'observation de figures et l'aggrégation de figures. Ces thèmes illustrent deux approches différentes au problème de la réutilisation des occurrences locales.

La première approche consiste à généraliser l'occurrence pour fusionner les préoccupations : celles-ci peuvent cohabiter dans l'implémentation du motif sans être totalement assimilées l'une à l'autre. Cependant cette approche entraîne l'invasion des classes impliquées dans le motif par toutes les préoccupations du motif.

La deuxième approche utilise classiquement l'héritage. Le risque est alors de privilégier la réutilisation au détriment des autres aspects de l'héritage. Ceci forme des hiérarchies de classes et d'interfaces complexes et en partie inutiles. Nous avons vu en particulier que cet usage peut dériver vers une implémentation indirecte d'interface et la création de relations implicites, donc fragiles, entre interfaces et classes.

4.4 Bilan

À travers l'étude de cas du cadriciel JHotDraw, nous illustrons les effets d'une forte densité de motifs tant sur l'abstraction que sur l'implémentation du programme.

Pour cette étude, nous avons mis en place une démarche ad hoc : l'identification d'un noyau de JHotDraw et son extension par de nouvelles préoccupations. Cette démarche nous permet d'expliquer les choix de conception et d'implémentation liées aux extensions non-modulaires des préoccupations. Or ce lien entre extensions et modifications du code est justement perdu dans l'implémentation. Notre démarche est donc aussi une illustration par l'exemple de la perte de traçabilité des choix de conception en rapport avec les motifs.

4.4.1 Pouvoir d'abstraction des motifs

En décrivant l'implication des motifs dans JHotDraw, nous montrons comment une documentation synthétique peut se construire sur la base des motifs. L'abstraction permise par les motifs permet d'expliquer efficacement des synergies (à la [Riehle, 1997]). Le cas des entités `Handle`, `Figure` et `Tool` via les motifs STATE, ADAPTER et FACTORY METHOD illustre une telle synergie.

Plus la densité est forte, plus il est possible d'expliquer l'implémentation du programme par des motifs. Nous distinguons deux usages caractéristiques des motifs à cet effet : la description d'une collaboration ou d'une structure entre les classes du programme et la description d'une extension du programme.

Mais la description permise par les motifs n'est pas exhaustive. Elle manque parfois d'un niveau suffisant de détail pour comprendre les choix de conception. Dans le cadre de la préoccupation d'invalidation, les interactions entre les motifs Composite, Decorator et Observer (et sa transformation en Chain of Responsibility) illustre la nécessité de spécifications détaillées.

Partant de ce constat sur l'abstraction possible avec les motifs, nous définissons deux métriques de densité : la métrique de densité de motifs évalue la maturité du code selon la part de fonctionnement assurée par les motifs ; la métrique de densité de rôles (ou score d'implication d'une classe) identifie les classes essentielles pour l'extension et la collaboration, appelées entités. Ces métriques servent d'indicateur pour exploiter ce pouvoir d'abstraction mais ne mesure pas la complexité ou la qualité du code.

Au final, nous soulignons qu'une forte densité de motifs est un indice quant au nombre de préoccupations du programme supportées par des motifs.

4.4.2 Impact des motifs sur l'implémentation objet

Une analyse approfondie de la conception et de l'implémentation est nécessaire pour comprendre l'impact des motifs. En particulier nous examinons quel est l'impact de l'ajout (ou du retrait) d'un motif dans le cadre d'une forte densité.

L'impact des motifs sur l'implémentation est généralement invasif : le code des motifs doit être inséré dans les classes. Nous montrons qu'il peut aussi être transversal : l'implémentation d'un motif dans une classe `A` peut entraîner des modifications sur un motif dans une classe `B`. Nous avons vu deux cas : une variation interne au motif par utilisation d'un autre motif ; mais aussi la transformation d'un motif Observer en Chain of Responsibility due aux interactions avec d'autres motifs.

Mais nous montrons qu'une forte densité offre aussi des opportunités pour la simplification et la réutilisation du code des motifs. Le motif Singleton est l'exemple type de simplification des implémentations. La réutilisation concerne des motifs à l'implémentation plus lourde, comme Observer et Composite. Il s'agit dans tous les cas de réutiliser les occurrences existantes dans le code (et non des implémentations génériques des motifs).

Cependant la réutilisation (ou la simplification) par opportunisme fragilise le code car elle implique un compromis entre les préoccupations supportées par la même occurrence. Les préoccupations doivent être fusionnées ou « détournées » par héritage. Nous examinons en particulier le cas du motif Composite dans `CompositeFigure` et `Drawing`, qui montre une copie indirecte, donc fragile, de l'interface de l'un par l'autre.

4.4.3 Conclusion

La densité de motifs peut donc être utilisée comme un indicateur abstrait de la maturité du code et du potentiel d'abstraction (pour la compréhension, la documentation) lié aux motifs. Mais nous montrons aussi par l'exemple les effets négatifs d'une forte densité sur l'implémentation. Les causes sont, comme indiquées à la section 2.2.2, avant tout la non-modularité des

implémentations des motifs et l'absence, en Java, de mécanisme de réutilisation adapté. Nous avons caractérisé certaines de ces modifications comme transversales.

Dans ce chapitre les implémentations des motifs sont juxtaposées, adaptées et réutilisées au cas par cas. À partir de ces observations, il est difficile de décrire ce qu'est une composition de motifs. C'est pourquoi nous nous attaquons dans le chapitre 5 à la modularisation des motifs par les aspects avant d'étudier la composition.

Chapitre 5

Implémentation des motifs de conception avec AspectJ

Sommaire

L'aspectisation des motifs de conception implique la transformation des solutions objets par les mécanismes aspects. Deux approches se distinguent suivant le degré de transformation atteint. L'approche idiomatique transforme partiellement la solution et modularise les autres éléments du motif dans un aspect : nous décrivons cette approche avec les idiomes propres à AspectJ.

L'approche par aspects capture le problème visé par le patron dans une nouvelle solution, indépendante du motif objet initial. Nous explorons les deux cas les plus probants, Decorator et Visitor, tout en soulignant les limites de l'intégration d'AspectJ aux objets.

Une grille de lecture de l'implémentation des motifs du *GoF* avec les mécanismes aspects permet une vision critique de ces deux approches.

5.1 Introduction

L'utilisation des nouveaux mécanismes aspects pour la modularisation des motifs transforme la solution objet initiale. Nous distinguons deux approches. L'approche idiomatique s'inspire de la solution objet initiale pour en extraire les éléments dispersés et les modulariser. L'approche par aspects propose une nouvelle solution au problème initial par un mécanisme différent.

Nous présentons plusieurs exemples de ces deux approches ainsi que le processus de transformation idiomatique d'une solution objet en aspect. Nous avons mené une étude extensive des solutions de [Hannemann et Kiczales, 2002] afin de cerner l'utilisation des différents mécanismes d'AspectJ et la part des motifs directement aspectisés par rapport aux motifs idiomatiques.

Cette étude montre les bénéfices d'AspectJ pour la modularisation et l'adaptation grâce aux liaisons tardives des coupes, mais aussi ses limites dans la définition des instances et des relations entre instances, lesquelles passent par des idiomes.

Pour détailler l'analyse de chaque motif, nous utilisons les termes suivant : une *propriété* est une spécification initiale du patron. Une *solution** est la spécialisation du motif pour un langage donné, utilisant donc les mécanismes et les idiomes propres à ce langage. Toute nouvelle solution doit donc satisfaire aux propriétés du patron. Une *caractéristique* est un élément spécifique à l'implémentation du motif : suivant le langage, certaines solutions implémentent ou non ces caractéristiques.

5.2 Solution idiomatique pour l'implémentation des motifs avec AspectJ

Un *idiome**, en matière de programmation, désigne une solution spécifique du langage face à un besoin courant d'implémentation. La différence par rapport aux patrons se fait sur l'abstraction du problème – proche de l'implémentation – et la spécificité de la solution – usage de mécanismes propre au langage.

La solution idiomatique d'un motif désigne donc une façon d'implémenter ce motif spécifique au langage visé. Cependant, dans le cadre de notre thèse et dans le contexte d'AspectJ, nous donnons un sens plus particulier à cette expression : la solution idiomatique désigne l'ensemble des idiomes du langage AspectJ pour une implémentation de la solution objet du motif. Une telle solution s'intéresse donc à retranscrire plus ou moins fidèlement la conception objet dans un aspect, contrairement à une solution attaquant directement le problème.

Le concept de solution idiomatique est intéressant car nous avons remarqué que de nombreux motifs dans Hannemann *et al.* présentaient de tels idiomes, ce qui marque leur caractère récurrent. Par ailleurs, plusieurs solutions idiomatiques sont possibles pour un même motif. Nous exposons donc ces idiomes à travers plusieurs solutions de motifs. Plus généralement, ces idiomes permettent l'implémentation d'une conception objet dans AspectJ.

5.2.1 Comparaison des solutions pour le motif Observer

Comme indiqué au chapitre 3 (section 3.2.2 page 42), le motif Observer est une cible de choix pour tester les solutions des aspects. La solution objet crée une dépendance du rôle sujet au rôle observateur (alors que la dépendance « fonctionnelle » est l'inverse – section 2.2.2.2 page 19) et impacte de façon invasive l'implémentation du sujet. De plus les notifications du rôle sujet au rôle observateur sont souvent dispersées et mélangées au code. La solution objet vise un

compromis de modularité en utilisant des dépendances abstraites (référence à une interface et notification abstraite). La solution aspect peut améliorer la modularité en inversant ces dépendances et modularisant les notifications : intuitivement, l'observateur est un aspect observant des sujets.

5.2.1.1 Caractéristiques du motif Observer

L'implémentation du motif Observer demande de considérer et raffiner plusieurs caractéristiques :

1. la détermination des classes jouant les rôles de sujet et d'observateur ;
2. la relation et le mécanisme de diffusion de sujets à observateurs, en particulier suivant la cardinalité : un sujet peut avoir un ou plusieurs observateurs ;
3. la définition des points de notification ;
4. le traitement par les observateurs des notifications ;
5. le passage de contexte à chaque notification, qui impacte la notification et son traitement : par exemple un sujet peut être passé comme paramètre de la notification pour permettre à l'observateur de distinguer entre plusieurs sujets ;
6. l'enregistrement (et le désenregistrement) des relations entre sujets et observateurs.

Trois des caractéristiques ci-dessus sont transformés par les mécanismes standards de la programmation par aspects. Elles constituent le squelette générique que l'on retrouve dans toute implémentation aspect du motif Observer (exemple 5.1). Les notifications et leur traitement (points 3 et 4) sont implémentés espectivement par une coupe (ligne 2) et une action (lignes 3 et 4). Les coupes permettent aussi le passage de contexte (point 5 et ligne 3). Les autres caractéristiques n'ont pas de correspondances bien établies dans les mécanismes des aspects. Elles sont donc tributaires du langage d'aspect choisi et en particulier de son intégration avec le langage sous-jacent.

Exemple 5.1 – Squelette d'une implémentation du motif Observer avec AspectJ

```
1 public aspect AnObserver {
2   pointcut notification(Subject s): ... ;
3   after(Subject subject): notification(subject) {
4     updateObserver(subject); }
5 }
```

La définition de la relation (points 2 et 6) fait partie de ces caractéristiques tributaires du langage. Intuitivement, l'observateur coupe les sujets dont il dépend et ignore les autres. L'aspect peut jouer directement ce rôle observateur si il est capable de définir la relation discriminant ses propres sujets. Sinon l'aspect joue le rôle de relai en capturant les notifications avant de les diffuser aux vrais observateurs.

Les mécanismes d'AspectJ permettent de discriminer la relation au sujet dans certains cas particuliers seulement. Dans le cas général, il faut définir l'aspect comme un relai d'observation. Les trois solutions Observer présentées ci-dessous illustrent ces choix et leurs implémentations.

5.2.1.2 Solution centralisée

La solution centralisée est un modèle où la plupart des éléments sont définis et instanciés dans une instance singleton de l'aspect. Pour chaque occurrence du motif OBSERVER, une seule instance d'aspect est donc définie et créée pour gérer toutes les relations de sujet à observateur. Elle sert donc de relai d'observation. Cette solution est celle présentée dans [Hannemann et Kiczales, 2002] et dans le source 5.1.

Code source 5.1 – Motif OBSERVER : solution centralisée réutilisable

```
1  public abstract aspect ObserverProtocol {
2    protected interface Subject  { }
3    protected interface Observer { }
4
5    private WeakHashMap perSubjectObservers;
6    protected List getObservers(Subject subject) {
7      if (perSubjectObservers == null) {
8        perSubjectObservers = new WeakHashMap(); }
9      List observers = (List)perSubjectObservers.get(subject);
10     if ( observers == null ) {
11       observers = new LinkedList();
12       perSubjectObservers.put(subject, observers); }
13     return observers;
14   }
15
16   public void addObserver(Subject subject, Observer observer) {
17     getObservers(subject).add(observer); }
18   public void removeObserver(Subject subject, Observer observer) {
19     getObservers(subject).remove(observer); }
20
21   protected abstract pointcut subjectChange(Subject subject);
22   after(Subject subject): subjectChange(subject) {
23     Iterator iter = getObservers(subject).iterator();
24     while ( iter.hasNext() )
25       updateObserver(subject, ((Observer)iter.next()));
26   }
27   protected abstract void updateObserver(Subject s, Observer o);
28 }
```

La relation des sujets aux observateurs est implémentée par une table de hâchage (ligne 5), où chaque sujet est une clé d'accès à sa liste d'observateurs (lignes 6–14). Cette structure est complètement encapsulée puisque privée et gérée par des `WeakReference`. Ainsi les sujets recyclables sont traités par le ramasse-miettes sans intervention de l'utilisateur. En contrepartie, l'utilisation d'une table de hâchage expose à des problèmes de performance avec des paramètres inadéquats.

Le processus d'enregistrement est centralisé par les méthodes `addObserver(Subject, Observer)` et `removeObserver(Subject, Observer)` (lignes 16–19). Une propriété de cette approche est d'offrir une interface publique et sans conflit puisque encapsulée dans l'aspect. Cette interface d'enregistrement est indépendante de l'utilisation de la table de hâchage, puisqu'il en est fait

abstraction via la méthode `getObservers(Subject)`.

Dans un aspect relai, l'action suivant une coupe de notification consiste à notifier chaque observateur (lignes 22–26).

Code source 5.2 – Motif Observer : spécialisation de la solution centralisée

```
1 public aspect FigureObserver extends ObserverProtocol{
2   declare parents: Figure  implements Subject;
3   declare parents: Drawing implements Observer;
4   protected pointcut subjectChange(Subject subject):
5     call(void Figure.move(..)) && target(subject);
6   protected void updateObserver(Subject s, Observer o) {
7     ((Drawing) o).invalidated((Figure) s); }
8 }
```

Abstraction et réutilisation. La propriété la plus intéressante de cette solution est sa réutilisabilité. Pour cela plusieurs caractéristiques dans la définition sont abstraites : les rôles du motif sont déclarés par des interfaces Java (lignes 2 et 3) ; les points de notifications sont représentés par une coupe abstraite (ligne 21) spécialisée dans un autre aspect (source 5.2, lignes 4–5) ; le traitement des notifications est centralisé et abstrait (ligne 27). La définition du traitement se fait dans l'aspect, au prix cependant du transtypage (source 5.2, lignes 6–7).

Cette propriété est liée à deux mécanismes : les interfaces Java et le mécanisme d'extension d'AspectJ. L'utilisation des interfaces Java permet d'abstraire les rôles du motif dans la relation, le passage de contexte, la notification des observateurs. L'extension des aspects se base sur la définition d'un aspect abstrait définissant les éléments réutilisables (relation, enregistrement, relai) et déclarant les méthodes et coupes abstraites (notifications et traitement). Un aspect abstrait ne peut être utilisé que par extension dans un aspect concret. Par principe celui-ci représente une seule occurrence du motif. Toute occurrence du motif est donc indépendante des autres puisqu'isolée dans son propre aspect (source 5.2). L'encapsulation et la centralisation des caractéristiques rendent impossibles les conflits en présence de multiples occurrences du motif.

5.2.1.3 Solution par déclaration intertype

La solution par déclaration intertype (DIT) utilise ce mécanisme propre à AspectJ (section 3.1.4.2 page 37). Elle est présentée dans le source 5.3.

Contrairement à la solution centralisée, la relation de chaque sujet à sa liste d'observateurs est directement défini dans le sujet (ligne 3). L'utilisation du modifieur `private` (relatif à l'aspect et non à la classe visée) garantit que cette liste n'est visible que de l'aspect, éliminant les risques de conflit avec d'autres attributs de la classe. Cette structure statique est équivalente à la solution objet classique du motif Observer.

L'exemple montre l'utilisation d'une coupe-action pour enregistrer la relation d'un sujet et de son observateur (lignes 8–10). Bien qu'il soit possible d'introduire l'interface pour l'enregistrement avec une déclaration intertype, ceci minimise le risque de conflit.

L'aspect a un comportement de relai. Grâce au contexte de la coupe (ligne 13), il récupère la liste d'observateurs du sujet et diffuse la notification (lignes 14–15).

Une différence notable par rapport à la solution centralisée est que le traitement de la notification est laissé à l'observateur et non à l'aspect (lignes 6 et 15). Ceci permet la redéfinition, élimine (en partie) le transtypage et rétablit le polymorphisme d'héritage (source 5.4, lignes 10–11).

Code source 5.3 – Motif Observer : solution réutilisable par déclaration intertype

```
1  public abstract aspect OberverProtocol {
2    public interface Subject { }
3    private Vector Subject.observers = new Vector();
4    private  Vector Subject.getObservers() {return observers;}
5    public interface Observer {
6      public void update(Subject s); }
7
8    abstract pointcut addObserver(Observer o, Subject s);
9    after(Observer o, Subject s): addObserver(o, s) {
10     s.observers.addElement(o); }
11
12   abstract pointcut subjectChanges(Subject s);
13   after(Subject s): subjectChanges(s) {
14     for (int i = 0; i < s.getObservers().size(); i++)
15       ((Observer)s.getObservers().elementAt(i)).update(s); }
16 }
```

Réutilisation. La réutilisation de cette solution est liée aux mêmes mécanismes d'extension et d'interfaces (source 5.4). Mais elle exploite en particulier la combinaison des interfaces avec les déclarations intertypes, ce qui est une forme d'héritage transversal de la structure.

Code source 5.4 – Motif Observer : spécialisation de la solution DIT

```
1  public aspect FigureObserver extends ObserverProtocol{
2    declare parents: Figure  implements Subject;
3    declare parents: Drawing implements Observer;
4    protected pointcut subjectChange(Subject subject):
5      call(void Figure.move(..)) && target(subject);
6    pointcut addObserver(Observer o, Subject s):
7      call(void Drawing.addFigure(..)) && target(o) && args(s);
8  }
9  public class Drawing { // Observer dans FigureObserver
10   public void update(Subject s) {
11     invalidated((Figure) s); }
12 }
```

5.2.1.4 Solution par instance d'aspect liée

Cette solution est intéressante pour sa simplicité d'implémentation dans le cas particulier où la relation de sujet à observateur est « bijective » – c'est-à-dire chaque sujet est lié à un seul observateur et réciproquement. Il est alors possible d'utiliser le mode d'instantiation `perthis` d'AspectJ au lieu du mode `singleton` par défaut (section 3.1.4.1 page 36). Dans ce mode, une instance d'aspect est créée automatiquement pour chaque sujet : l'aspect peut jouer directement le rôle d'observateur pour le sujet lié.

L'implémentation en est simplifiée (source 5.5). La relation est déclarée par l'aspect (ligne 1), ce qui, par rapport aux solutions précédentes, élimine la définition d'une structure ainsi que la gestion de cette structure.

Le traitement de la notification est directement effectué dans l'action, sans relai de diffusion (lignes 4–5).

Code source 5.5 – Motif Observer : solution par instance liée

```
1 public aspect FigureObserver perthis(execution( Figure+.new(..)) ){
2   pointcut subjectChange(Figure figure):
3     call(void Figure.move(..)) && target(figure);
4   after(Figure figure): subjectChange(figure) {
5     invalidated(figure); }
6 }
```

Deux remarques montrent les limites de ce cas particulier :

- premièrement, si un sujet a plusieurs observateurs, la solution peut être aménagée pour définir la liste d'observateurs – cependant ceci nous ramène aux solutions précédentes d'aspect relai sans bénéfice ;
- deuxièmement, il est strictement impossible pour l'aspect observateur d'avoir plusieurs sujets ; en effet le mode d'instantiation `perthis` restreint aussi la portée de la coupe : seuls les points de jonction exécutés dans l'instance sujet lié sont notifiés (ce mode est détaillé en section 3.1.4.1 page 36).

Cette dernière remarque demande de vérifier deux défauts potentiels dans la définition de l'aspect observateur :

1. la coupe paramétrant l'instantiation `perthis(..)` doit être déclenchée avant toute coupe de notification – sinon une erreur à l'exécution sera déclenchée pour défaut d'instance ;
2. les coupes de notification ne peuvent capturer que des événements internes – on préférera par exemple la primitive de coupe `execution` à `call`, cette dernière pouvant faire référence à des points de jonction externes à l'instance qui ne seraient jamais capturés.

Enfin la simplicité et la spécificité de cette solution rendent la définition d'une abstraction réutilisable triviale et peu intéressante. Nous considérons cette solution du motif Observer plus comme une spécificité d'AspectJ que comme une solution générique.

5.2.1.5 Bilan

Nous constatons que la plupart des caractéristiques du motif Observer peuvent être définies suivant deux styles différents : centralisé dans l'aspect ou distribué avec des déclarations inter-

types. La relation peut être implémentée par une table de hâchage ou bien par une structure intertype. L'enregistrement peut être centralisé dans une interface de l'aspect ou une coupe-action, ou encore dans une interface intertype. Il est possible de mélanger ces styles – par exemple, utiliser une structure intertype avec une interface centralisée (source 5.3).

La réutilisation est possible grâce à l'abstraction par les interfaces et à l'extension des aspects.

Les critères de choix entre ces solutions dépendent alors des spécifications et contraintes de l'occurrence visée. On peut choisir une interface publique dans l'aspect ou au contraire cacher l'aspect via une coupe ou une interface introduite dans les classes. L'usage d'une table de hâchage induit un surcoût d'exécution et, sous certaines conditions, peut être inefficace. Les déclarations intertypes publiques peuvent entraîner des conflits dans les classes. Enfin l'adaptation du motif dépend aussi de la part déléguée par l'aspect aux classes (par des méthodes ou des déclarations intertypes publiques).

Cependant le cas général du motif Observer, où de multiples sujets sont en relation avec de multiples observateurs, ne peut être implémenté que par un aspect relai, capturant tous les points de notification avant de les diffuser aux observateurs concernés. L'aspect doit définir et gérer cette relation de façon ad hoc. C'est un point fondamental des idiomes d'AspectJ pour l'implémentation des motifs.

5.2.2 Idiomes d'implémentation d'un motif en AspectJ

Les solutions pour le motif Observer font la démonstration des idiomes AspectJ propres à l'implémentation des motifs. Nous faisons un exposé plus générique de ces idiomes à partir des rubriques du *GoF* : **Participants**, **Structure**, **Collaborations**. L'objectif premier de cette démarche et de ces idiomes est la modularisation du motif dans un aspect.

Comme indiqué dans le bilan 5.2.1.5, l'implémentation d'une caractéristique avec AspectJ peut parfois être réalisée de différentes manières. D'une part les déclarations intertypes distribuent les modifications sur différentes classes. D'autre part les actions déclenchées par les coupes s'exécutent dans le contexte centralisé de l'aspect. Ceci a des conséquences sur l'usage des motifs, les risques de conflit et principalement sur l'implémentation de la structure d'un motif. Ces approches ont des propriétés différentes mais ne sont pas forcément incompatibles, comme montré dans les sections suivantes.

5.2.2.1 Participants

Les participants désignent les classes et les objets impliqués dans le motif. Un participant joue dans la solution un rôle spécifique, dont le nom indique d'ailleurs la responsabilité. Tracer les participants est fondamental pour comprendre le lien entre le motif et l'occurrence.

Les interfaces Java sont utilisées dans AspectJ pour représenter ces rôles et désigner via celles-ci les classes impliquées. Les interfaces Java, étant transversales à l'héritage, n'ont pas d'impact sur l'arbre d'héritage. Un rôle dans un motif peut donc être appliqué à n'importe quelle classe, soit avec une déclaration d'implémentation de l'interface par la classe, soit avec une déclaration intertype (`declare parents: AClass implements AnInterface`) dans l'aspect. Cet idiome des *interfaces-rôles*, appliqué systématiquement, permet de tracer facilement les classes participantes au motif.

Deux remarques empiriques peuvent être faites sur l'usage des interfaces Java dans le cadre d'AspectJ. D'une part ces interfaces-rôles sont souvent vides, sans déclaration de méthode, car

les collaborations entre motifs sont définies de façon différente. D'autre part le rôle Client, présent dans la description des patrons, est ignoré dans l'aspectisation des motifs car défini uniquement comme un utilisateur de l'interface publique du motif.

5.2.2.2 Structure

La structure d'un motif définit les relations d'héritage, d'agrégation et d'association entre les classes des participants.

L'implémentation des interfaces-rôles par les classes participantes remplace le sous-classage de l'héritage. L'usage des déclarations intertypes en combinaison avec les interfaces (voir section 3.1.4.2 et source 5.3, lignes 2 à 4) permet la réutilisation de code, en remplacement de l'héritage.

Deux idiomes différents sont disponibles pour définir une relation d'agrégation ou d'association. La référence au participant ciblé peut être introduite dans la classe source par une déclaration intertype (privée). C'est l'idiome de la *structure intertype*, calquant le modèle objet (section 5.2.1.3). Mais l'aspect (singleton) peut aussi définir une structure unique centralisant la relation de participant à participant : chaque source de la relation est une clé dans une table de hâchage vers sa ou ses cibles (section 5.2.1.2). C'est l'idiome de la *structure centrale*. L'exemple de la solution Observer par instance liée (section 5.2.1.4) montre que l'utilisation des modes d'instantiation d'AspectJ est aussi possible mais limitée par des contraintes sur les spécifications du motif (relation bijective).

5.2.2.3 Collaborations

Les collaborations des motifs désignent les activités internes, entre participants, aussi bien que l'usage externe par le client (via l'interface publique du motif). En particulier, les collaborations décrivent comment les responsabilités du motif sont distribuées entre les différents participants.

La transformation des collaborations par les mécanismes des aspects est celle impliquant en général le plus de changement. Une collaboration peut en effet être remplacée par un tandem coupe-action : la dépendance entre les participants est alors inversée, comme entre les rôles Sujet et Observateur du motif Observer (section 5.2.1). La collaboration est alors une réaction à la capture d'un point de jonction. Le tandem coupe-action est utilisé quand les points de dispersion sont nombreux. Cependant l'extension a posteriori d'une coupe-action (après définition de l'aspect) de l'application pose problème. La coupe peut ne pas être suffisamment générique pour capturer de nouveaux points. L'action d'un aspect ne peut être directement étendue. Elle peut faire appel à une méthode qui, à son tour, sera redéfinie.

Comme pour la structure des motifs, il est possible d'utiliser les déclarations intertypes. Les méthodes intertypes calquent la solution objet pour implémenter les collaborations. L'utilisation des méthodes intertypes autorise la redéfinition, donc l'extension et le polymorphisme d'héritage. Cependant il existe un risque de conflit avec d'autres méthodes du même nom si la déclaration de la méthode intertype est publique.

La définition d'une gestion des relations d'agrégation ou d'association illustre les choix et compromis liés à des mécanismes d'AspectJ. Une coupe-action permet de créer des relations entre objets de façon interne (par exemple l'association d'une instance Observateur en réaction à la création d'un nouveau Sujet). Une interface publique permet à l'utilisateur de gérer celles-ci de façon externe. Cette interface publique peut être définie soit dans les classes participantes avec

une déclaration intertype, soit dans l'aspect central. L'inconvénient d'une déclaration intertype publique de l'interface est le risque de conflit avec d'autres méthodes des classes. Ce risque n'existe pas avec une interface publique centralisée dans l'aspect : cet idiome de *l'interface centrale* est assimilé au motif FACADE.

5.2.2.4 Motif et occurrence

Les éléments d'implémentation ci-dessus permettent de localiser dans l'aspect les caractéristiques d'un motif. Ceci élimine le problème de traçabilité des éléments dispersés des motifs. De plus il est possible de définir, grâce au mécanisme d'extension d'AspectJ, une version abstraite et réutilisable pour certains motifs, tel les solutions centralisée (section 5.2.1.2) et par DIT (section 5.2.1.3) pour le motif OBSERVER. L'aspect abstrait définit une version traçable du motif générique, tandis que l'aspect concret définit une *occurrence** du motif.

5.2.2.5 Synthèse

En observant les implémentations de [Hannemann et Kiczales, 2002], nous pouvons dégager les caractéristiques d'une transformation idiomatique d'un motif avec AspectJ :

- la modularisation du motif dans son ensemble par un aspect singleton ;
- la déclaration des rôles par des interfaces Java internes à l'aspect ;
- la définition des relations entre rôles via des structures intertypes ou une table de hâchage centralisée ;
- la transformation des collaborations par des coupes-actions et des méthodes (intertypes ou non) ;
- la centralisation dans l'aspect des interfaces publiques du motif.

Le choix entre idiome de structure intertype ou idiome de structure centralisée a un impact fondamental sur l'implémentation de l'aspect : chaque idiome impose un style de code aux collaborations (qu'ils s'agissent de coupes-actions ou de méthodes). Cette transformation a pour but de modulariser un motif sans provoquer de conflit par des déclarations intertypes publiques. Néanmoins il est possible de mélanger déclarations intertypes et mécanismes centralisés dans un même aspect.

Nous remarquons enfin que la centralisation dans l'aspect est en opposition avec la distribution classique des responsabilités entre participants objets. Ceci se traduit par la difficulté à utiliser le polymorphisme d'héritage pour étendre un motif dans un aspect. À l'inverse les déclarations intertypes permettent ce polymorphisme d'héritage.

5.3 Exemples de motifs aspectisés

Les mécanismes de la programmation par aspects permettent de reconsidérer les solutions aux problèmes décrits par certains patrons de conception. À la différence d'une imitation idiomatique, la solution objet initiale est donc ignorée. Ces motifs aspectisés ont des propriétés différentes des solutions objets.

5.3.1 Motif Decorator

Les propriétés liées au motif DECORATOR sont simples mais intéressantes : il permet d'attacher *dynamiquement* des *responsabilités additionnelles* à un objet existant sans changer son

interface. Le motif DECORATOR est une alternative entre extension par héritage et extension par délégation. Comme avec l'héritage, l'extension est transparente pour l'objet décoré. Comme avec l'agrégation, le motif DECORATOR permet la composition récursive : il est possible d'appliquer récursivement un `Decorator` autour d'un autre `Decorator`.

Dans sa solution objet, le motif DECORATOR utilise l'héritage et la délégation. Une instance de classe `Decorator` se substitue et encapsule l'objet décoré. La classe `Decorator` hérite de la classe décorée pour permettre la subtitution et redéfinit chaque méthode héritée pour décorer ou au minimum déléguer l'appel à l'objet encapsulé (voir exemple ci-dessous). La délégation de toutes les méthodes de l'interface illustre le problème du surcoût d'implémentation (section 2.2.2.4 page 20). Enfin cette solution n'est effectivement appliquée qu'à partir du moment où la référence de l'instance `Decorator` remplace la référence à l'objet décoré. Pour utiliser cette solution, il faut donc définir quand et à quels endroits la substitution se fait. Au final la solution objet est lourde à implémenter mais bien modularisée dans une classe `Decorator`.

```
class BorderDecorator implements Figure { // sous-type pour substitution
  private Figure component; // objet decore
  void move(int x, int y) { // methode deleguee
    component.move(x, y); }
  void draw(Graphics g) { // methode decoree
    drawBorder(g);
    component.draw(g); }
}
```

Une implémentation idiomatique du motif DECORATOR en AspectJ n'apporte aucun bénéfice. Il faut définir la relation décorateur-décoré par un idiome de structure, intercepter chaque appel de méthode au décorateur et le rediriger sur l'objet décoré. Si l'interception de l'appel peut être faite par une coupe-action, la redirection, comme dans la solution objet, doit être définie pour chaque méthode de l'interface. Cette solution ne résoud pas non plus le problème de surcoût d'implémentation.

Cependant la programmation par aspects permet des extensions transparentes à la manière du motif DECORATOR. Nous explorons dans les deux sections suivantes les solutions aspects au motif DECORATOR.

5.3.1.1 Motif Decorator statique

La solution aspectisée du motif DECORATOR est triviale avec les mécanismes basiques de la programmation par aspects. Le code source 5.6 donne un exemple d'une occurrence de DECORATOR d'après [Hannemann et Kiczales, 2002]. Chaque méthode décorée est interceptée par une coupe (lignes 2–3) pour être modifiée par une action (lignes 4 à 6). Cette solution permet de composer statiquement plusieurs occurrences différentes de DECORATOR. L'ordre de composition est définie par précédence.

Cette solution apporte plusieurs avantages par rapport à la solution objet :

1. seules les méthodes à décorer sont modifiées avec une action ; il n'y a pas de surcoût d'implémentation des méthodes de simple délégation ;
2. conséquemment, il n'y a pas d'indirection pour les méthodes non-décorées, réduisant le coût d'exécution ;
3. le motif est appliqué statiquement aux classes décorées ; il n'y a ni instantiation ni remplacement de référence.

Code source 5.6 – Motif DECORATOR : occurrence statique

```
1 public aspect BorderDecorator {
2  pointcut draw(Graphics g):
3   execution(void Figure.draw(Graphics)) && args(g) ;
4  void around(Graphics g): draw(g) {
5   drawBorder(g);
6   proceed(g); }  // appel de la methode decoree
7 }
```

Cependant ce dernier point est en contradiction avec la propriété d'attachement dynamique du motif DECORATOR. Cette solution est plus proche de l'héritage statique. Par ailleurs deux problèmes de composition dynamique apparaissent :

1. il n'est pas possible de composer récursivement plusieurs instances d'une même occurrence ;
2. il n'est pas possible de changer dynamiquement l'ordre de composition entre occurrences.

5.3.1.2 Motif Decorator dynamique

Nous proposons une version dynamique du motif DECORATOR (code source 5.7). L'élément principal est l'association d'un état dans l'aspect à chaque instance décorée. Nous utilisons pour cela l'idiome de relation bijective déjà vue avec l'occurrence de OBSERVER par instance liée (section 5.2.1.4).

Code source 5.7 – Motif DECORATOR : occurrence dynamique

```
1 public aspect BorderDecorator perthis(draw(Graphics)){
2  private boolean active = false;
3  public void decorate(){ this.active=true; }
4  public void strip(){ this.active=false; }
5  pointcut draw(Graphics g):
6   execution(void Figure.draw(Graphics)) && args(g) ;
7  void around(Graphics g): draw(g) {
8   if( this.active )
9     drawBorder(g); // decoration seulement si active
10  proceed(g); }
11 }
12 // Usage
13 BracketDecorator.aspectOf(aConcreteOutput).decorate();
```

Une variable `active` (ligne 2) est définie pour chaque instance de l'aspect `BorderDecorator`. Or pour chaque instance de `Figure`, une instance `BorderDecorator` est associée par la déclaration `perthis(...)` (ligne 1) et la coupe `draw(...)` (lignes 5–6) : une variable `active` est donc associée à chaque `Figure` pour indiquer si elle doit être décorée ou non. Dans chaque action de décoration, un test sur la variable `active` (ligne 8) contrôle l'exécution de la modification (ligne 9). La méthode décorée est toujours appelée avec `proceed` pour permettre son exécution (ligne

10). Les méthodes `decorate` et `strip` de `BorderDecorator` permettent l'application dynamique du comportement.

Cependant cette solution est aussi sujette à critique : l'instance `Decorator`, même inactive, est toujours présente d'où un surcoût à l'exécution. De plus cette solution ne résoud pas les problèmes de composition récursive et d'ordre dynamique.

5.3.1.3 Synthèse sur le Decorator aspectisé

Les solutions du motif DECORATOR proposées ci-dessus illustrent une conception différente des solutions objets. Contrairement à l'implémentation idiomatique où l'aspect est omnipotent et joue plusieurs rôles, l'aspect joue l'unique rôle du décorateur, ce qui simplifie son implémentation. Cette simplification est aussi due au mécanisme `around` d'AspectJ. Cependant les propriétés de composition dynamique sont difficiles voire impossibles à émuler. Nous notons enfin que la solution dynamique (section 5.3.1.2) utilise l'idiome d'état de la solution OBSERVER par instance liée (section 5.2.1.4).

5.3.2 Motif Visitor

Le motif VISITOR permet de définir une opération pour une structure de données composée de classes hétérogènes tout en spécialisant facilement cette opération pour chaque classe. Chaque opération est encapsulée dans une classe `Visitor`. Ceci permet l'ajout modulaire d'une nouvelle opération pour la même structure de données.

La caractéristique principale de la solution objet est l'introduction d'un mécanisme de double envoi entre les classes de données et la classe `Visitor`. Chaque objet de la structure appelle l'instance `Visitor` avec une méthode dédiée à sa classe. L'interface de `Visitor` déclare donc une méthode pour chaque classe de données. La classe `Visitor` définit une fonction totale sur la structure en implémentant toutes ces méthodes. Les classes de données dépendent seulement de l'interface `Visitor`, ce qui permet d'ajouter une opération avec la même interface de façon modulaire.

Cependant l'implémentation du double envoi a une contrepartie problématique : l'interface `Visitor` est sensible aux classes de la structure de données. L'ajout d'une nouvelle classe de données a un impact sur toutes les classes `Visitor`. Ainsi le choix du motif VISITOR est souvent conditionné par le besoin d'un ensemble dynamique d'opérations et la nécessité d'un ensemble statique de classes de données.

5.3.2.1 Motif Visitor idiomatique

La solution AspectJ proposée dans [Hannemann et Kiczales, 2002] (source 5.8) copie les caractéristiques de la solution objet. Les interfaces-rôles sont définies pour une structure avec deux types différents de nœuds (`Node` et `Leaf`, lignes 2–4) ainsi que pour l'opération `Visitor` (ligne 6). La caractéristique intéressante de cette solution est qu'elle généralise l'implémentation du mécanisme de double envoi dans les classes de données avec des déclarations intertypes (lignes 12–16). Il n'y a pas d'impact sur le code source des classes de données. L'aspect `VisitorProtocol` encapsule ces définitions. Cette solution a les caractéristiques d'une implémentation idiomatique.

Cette solution est aussi intéressante car réutilisable. Le code source 5.9 montre la définition d'une occurrence par implémentation de l'interface `Visitor`.

Code source 5.8 – Motif VISITOR : solution idiomatique réutilisable

```
1  public abstract aspect VisitorProtocol {
2     public interface VisitableNode {}
3     protected interface Node extends VisitableNode {}
4     protected interface Leaf extends VisitableNode {}
5
6     public interface Visitor {
7        public void visitNode(VisitableNode node);
8        public void visitLeaf(VisitableNode node);
9        public String report(); // methode d'acces au resultat
10    }
11
12    public void VisitableNode.accept(Visitor visitor) {}
13    public void Node.accept(Visitor visitor) {
14       visitor.visitNode(this); }
15    public void Leaf.accept(Visitor visitor) {
16       visitor.visitLeaf(this); }
17 }
```

Cependant cette solution idiomatique souffre du même défaut de sensibilité aux classes de la structure de données que la solution objet. Le code source 5.8 n'est réutilisable que pour les structures disposant de deux types de nœuds. En cas de structure différente ou de changement dans la structure, l'interface `Visitor` doit être adaptée – ce qui a un impact sur toutes les occurrences de ce VISITOR.

5.3.2.2 Motif Visitor générique

Nous proposons une nouvelle solution palliant au défaut de sensibilité. Ce défaut est issu de la nécessité de déclarer dans l'interface générique `Visitor` chaque classe spécifique de la structure. Nous éliminons cette nécessité en remplaçant le double envoi par une sélection dynamique avec des coupes. Celles-ci utilisent les prédicats de filtrage d'AspectJ pour sélectionner l'action correcte d'après les classes de données. Cette solution émule un mécanisme d'envoi multiple.

En conséquence la définition et le protocole de collaborations du motif sont simplifiés. Le code source 5.10 déclare la part générique du motif. L'aspect définit directement l'opération `Visitor`. La méthode `visit` est définie dans l'interface de l'aspect (ligne 3). La déclaration de la structure se limite à un type générique `VisitableNode` (ligne 2). Le protocole voit la disparition de la méthode `accept` de l'interface de `VisitableNode`. Le protocole implique l'appel direct de la méthode `visit` en passant les objets de la structure en paramètre. Cet appel est en fait capturé par la coupe `visit` (lignes 4–5).

Le code source 5.11 montre la spécialisation du motif VISITOR générique. L'occurrence `CountingVisitor` compte le nombre de feuilles `TestLeaf` dans une structure arborescente à deux classes (`TestLeaf` et `TestGroup extends TestLeaf`, non montrées).

Chaque classe de données est capturée par un prédicat `args` composé avec la coupe `visit`. La signature de chaque action permet de discriminer les classes pour sélectionner l'action la plus spécifique (`TestLeaf` ligne 13 et `TestGroup` ligne 16). Le traitement d'une nouvelle classe passe

Code source 5.9 – Motif VISITOR : spécialisation de la solution idiomatique

```
1  public class CountVisitor implements VisitorProtocol.Visitor {
2    protected int count = 0;
3    public void visitNode(VisitorProtocol.VisitableNode node) {
4      if (node instanceof BinaryTreeNode) {
5        BinaryTreeNode rnode = (BinaryTreeNode) node;
6        rnode.left.accept(this);
7        rnode.right.accept(this); }
8    }
9    public void visitLeaf(VisitorProtocol.VisitableNode node) {
10     if (node instanceof BinaryTreeLeaf)
11       count += 1; }
12   public String report() {
13     return "Number of leaves " + sum; }
14 }
```

Code source 5.10 – Motif VISITOR générique

```
1  public abstract aspect Visitor {
2    public interface VisitableNode {}
3    public void visit(VisitableNode node){}
4    public pointcut visit():
5      execution(void Visitor.visit(VisitableNode));
6  }
```

simplement par la définition de la coupe et de l'action correspondante (par exemple, lignes 13 à 15) et, si nécessaire, la déclaration d'implémentation de l'interface `VisitableNode` ligne 11 (si cette déclaration n'est pas héritée via une autre interface). Cette solution n'est donc pas sensible, dans sa partie générique, à la structure visée. Chaque occurrence de VISITOR est libre de traiter chaque type de la structure.

L'usage de cette solution est aussi illustré lignes 20 à 23. Les méthodes `instance` (statique) et `init` (lignes 3 à 5) facilitent cet usage en simplifiant l'accès à l'instance `Visitor` et sa réinitialisation. Ces méthodes et cet usage soulignent cependant le défaut potentiel de cette solution pour les occurrences à état : du fait de la nature singleton de l'aspect, une seule instance de l'occurrence `CountingVisitor` est créée. Ceci présente un risque en cas d'oubli de réinitialisation lors d'un usage consécutif de l'instance ou dans un contexte concurrent où deux processus peuvent exploiter l'état en parallèle.

La solution présentée ci-dessus montre une version générique et réutilisable du motif VISITOR à fin de comparaison avec la solution idiomatique. Cependant l'intérêt du motif abstrait 5.10 est limité étant donné d'une part sa concision, d'autre part les pénalités qu'il implique dans la définition des occurrences de VISITOR. Ces pénalités sont la déclaration du lien à `VisitableNode` par les classes de données (ligne 11) et surtout le filtrage du type spécialisant `Visitor` dans la coupe générique `visit` (ici factorisée par la coupe `myvisit` ligne 12). Ce filtrage est nécessaire pour éviter que différentes occurrences VISITOR ciblant la même structure ne s'exécutent en même temps sur la même coupe générique `visit`.

Code source 5.11 – Motif VISITOR : spécialisation de la solution générique

```
1  aspect CountingVisitor extends Visitor {
2      // methodes d'usage
3      public static CountingVisitor instance() {
4         return CountingVisitor.aspectOf().init(); }
5      public CountingVisitor init(){ this.count = 0; return this; }
6
7      private int count = 0;
8      void incr(){ count++; }
9      public int report(){ return count; }
10
11     declare parents: TestLeaf implements VisitableNode;
12     pointcut myvisit(): visit() && this(CountingVisitor);
13     void around(TestLeaf l): myvisit() && args(l) {
14         System.out.println("Visiting a TestLeaf " + l);
15         this.incr(); }
16     void around(TestGroup c): myvisit() && args(c){
17         System.out.println("Visiting a TestGroup " + c); }
18 }
19 // Usage
20 CountingVisitor counter = CountingVisitor.instance();
21 for(VisitableNode aNode: nodes)
22     counter.visit(aNode);
23 System.out.println(counter.report());
```

En conclusion nous recommandons, sauf cas nécessaire, la définition directe du motif générique dans les occurrences plutôt que son extension. Nous notons aussi que cette solution est particulièrement concise et adaptée aux `Visitor` sans état, pour lesquels les problèmes de concurrence et d'initialisation ne se posent pas.

5.3.2.3 Motif Visitor avec état persistant

Il existe plusieurs solutions pour pallier au défaut de l'état singleton de la solution précédente. Elles consistent à associer un état au contexte d'exécution. Une solution peut exploiter par exemple le mode d'instantiation `percflow` d'AspectJ – qui crée une instance par flot de contrôle capturée – pour définir un état pour la durée de la traversée d'un `Visitor`. Il faut dans ce cas veiller à récupérer le résultat avant la sortie du flot de contrôle.

Nous présentons (code source 5.12) une solution plus générale basée sur l'association d'un objet et d'un aspect. Le rôle de l'objet n'est pas tant de stocker l'état que de limiter la portée de l'aspect.

La classe `CountingVisitor2` reprend le rôle précédent de `Visitor`, en particulier pour la déclaration de la méthode `visit`. L'aspect `CountingVisitor2Behavior` reprend la définition des coupes et des actions. La différence vient de l'association d'une instance d'aspect à chaque objet `CountingVisitor2` avec la déclaration `perthis` (ligne 7). L'instance d'aspect est limitée dans sa portée au seul objet auquel elle est liée.

La coupe `instancevisitor` et la définition de la variable d'instance `myVisitorInstance`

par une coupe-action constituent un *idiome d'association* (lignes 8 à 11) pour faciliter l'accès (ligne 17) à l'objet associé par `perthis`.

Code source 5.12 – Motif VISITOR : occurrence avec état persistant

```
1  public class CountingVisitor2 {
2    private int count = 0;
3    void incr(){ count++; }
4    public int report(){ return count; }
5    public void visit(TestLeaf test) {}
6  }
7  aspect CountingVisitor2Behavior perthis(instancevisitor()){
8    pointcut instancevisitor(): execution(CountingVisitor2.new(..));
9    private CountingVisitor2 myVisitorInstance;
10   after(CountingVisitor2 v): instancevisitor() && this(v) {
11     myVisitorInstance = v; }
12
13   pointcut visit():
14     execution( void CountingVisitor2.visit(TestLeaf) );
15   void around(TestLeaf l): visit() && args(l){
16     System.out.println("2> Visiting a TestLeaf " + l);
17     myVisitorInstance.incr(); }
18   void around(TestGroup c): visit() && args(c){
19     System.out.println("2> Visiting a TestGroup " + c); }
20 }
```

L'usage de cette solution est similaire à la précédente (code 5.11, lignes 20–23). Il faut créer un objet de la classe `CountingBehavior2` puis appeler la méthode `visit` en passant chaque objet de la structure de données. Le risque de réutiliser ou de partager une instance entre deux processus par inadvertance est réduit, même si de telles options sont toujours possibles.

5.3.2.4 Bilan sur le motif Visitor

Nous avons vu que la solution idiomatique du motif VISITOR, bien que facilitant la définition du mécanisme de double envoi, souffre du même défaut de sensibilité aux classes de la structure de données que la solution objet.

Nous proposons deux solutions basées sur une aspectisation plus poussée du motif. Ces deux solutions reposent sur l'utilisation des coupes et des prédicats de filtrage de type pour discriminer les classes de données et sélectionner les actions. La coupe assouplit la signature par rapport à une méthode normale : cette souplesse permet de séparer signature générique et signature de filtrage réelle, ce qui résoud le problème de sensibilité. Ce mécanisme peut être étendu à plusieurs paramètres.

La définition d'actions par rapport à un filtrage dynamique des paramètres est similaire à la surcharge. Ces solutions VISITOR permettent donc un polymorphisme ad hoc.

Cependant ces solutions montrent aussi qu'il est délicat de travailler avec AspectJ sur la portée des aspects. Ceci a des conséquences tant dans l'utilisation d'un motif générique que dans la définition d'un état. Dans le premier cas, il faut filtrer les coupes avec le type d'un aspect concret ; dans le second, veiller à ne pas partager un état singleton. Ce problème peut

être résolu d'une manière générale en associant à un aspect un objet qui en limitera la portée, mais au prix d'une complexité plus grande à l'implémentation.

5.4 Revue des motifs du *GoF* avec AspectJ

Nous avons illustré avec les mécanismes d'AspectJ les approches idiomatiques et par aspects pour l'implémentation des motifs. Nous regardons maintenant comment ces deux approches se partagent l'ensemble des motifs du *GoF* afin d'estimer leur importance relative.

Pour cette étude nous sommes partis du projet développé par [Hannemann et Kiczales, 2002] faisant la démonstration d'une implémentation pour chaque motif. Cette étude ne prend donc pas en compte les nouvelles solutions des motifs Observer, Decorator ou Visitor que nous avons exposées. Nous procédons ainsi de façon à révéler l'importance de certains idiomes dans Hannemann *et al.*.

Nous présentons une vue générale de l'ensemble des motifs dans le tableau 5.1. Ce tableau utilise deux critères qualitatifs – l'un décrivant l'implémentation, l'autre le degré d'abstraction et de réutilisation – pour classer les *solutions** sous AspectJ. Ce tableau montre en particulier l'importance des idiomes des interfaces-rôles et de la structure centrale pour la réutilisation.

5.4.1 Critères généraux pour la revue

Nous utilisons deux critères suffisamment généraux pour classer les solutions AspectJ des motifs : l'un est l'idiome ou le mécanisme prédominant à l'implémentation du motif ; l'autre est le niveau d'abstraction atteint par la solution, qui permet un regard critique sur la réutilisation possible.

5.4.1.1 Catégories d'implémentation

Nous distinguons trois catégories d'implémentation suivant l'idiome ou le mécanisme prédominant dans les solutions AspectJ proposées par Hannemann *et al.*.

Motif DIT (déclaration intertype) : la structure et les collaborations du motif sont définies par des déclarations intertypes dans un aspect

Motif aspectisé : l'aspect définit les collaborations mais pas de structure.

Motif centralisé : l'aspect utilise l'idiome de structure centrale et centralise les collaborations.

Ces catégories reflètent l'uniformité des solutions de Hannemann *et al.* dans l'utilisation des mécanismes : peu de motifs de la catégorie DIT utilisent des coupes, des actions, ou une interface centralisée ; à l'inverse les motifs centralisés définissent presque tous une interface publique centralisée, bien qu'ils n'utilisent pas systématiquement coupes et actions dans leurs collaborations. Les motifs aspectisés sont simplifiés par l'utilisation des coupes et des actions et comportent en conséquence peu ou pas d'idiomes. Il y a donc peu de mixité dans l'utilisation des mécanismes structurels et comportementaux. Quelques exceptions notables seront détaillées par la suite.

Une catégorie **Complément** regroupe les motifs pour lesquels AspectJ offre un support à l'implémentation du motif sans transformer la solution objet originale. L'aspect complémente la solution avec des méthodes d'aide mais le mécanisme utilisé par le motif (par exemple le clonage) est orthogonal aux aspects.

5.4.1.2 Niveau d'abstraction

Nous définissons cinq niveaux d'abstraction pour refléter la proximité de la solution aspect avec un (hypothétique) motif générique. L'objectif de ce critère est de hiérarchiser les solutions aspects par rapport à leur réutilisabilité.

Motif générique : l'aspect définit une solution générique du motif, c'est-à-dire adaptable à la plupart des situations décrites par le patron ;

Motif réutilisable : l'aspect définit une solution réutilisable du motif, tout en utilisant des choix de conception spécifiques à un cas particulier du patron ;

Occurrence extensible : l'aspect définit une solution spécifique à un problème de l'application mais peut être étendu de façon modulaire ;

Occurrence spécifique : l'aspect définit une solution spécifique à un problème de l'application ;

La différence principale est entre le niveau motif – caractérisé par un aspect abstrait réutilisable en AspectJ – et le niveau occurrence. Les distinctions entre sous-niveaux sont plus empiriques. En effet ce critère est influencé par la disparité des motifs de conception (section 2.2.1 page 15) : il est plus difficile de définir une solution générique pour un motif comme Observer, disposant de nombreuses variantes.

De même que pour la catégorie Complément, nous définissons un niveau **Orthogonal** pour classer certains motifs pour lesquels AspectJ ne change pas le mécanisme d'extension.

5.4.2 Tableau des solutions et analyse

Le tableau 5.1 présente les solutions de Hannemann *et al.* suivant les critères d'implémentation et d'abstraction. La lisibilité obtenue par ce moyen demande cependant une analyse plus fine pour en expliquer la répartition (section 5.4.2.1 et suivantes) ou commenter les cas particuliers (section 5.4.3).

5.4.2.1 Répartition des solutions

La petite taille de la catégorie des motifs aspectisés montre que peu de motifs sont complètement transformés par les mécanismes d'AspectJ. Le motif Decorator est discuté en section 5.3.1. Les motifs Proxy et State sont discutés en section 5.4.3.

Dichotomie entre motifs DIT et centralisés. Le caractère le plus évident dans ce tableau est la dichotomie entre motifs DIT et centralisés. Cette dichotomie est intéressante car elle est conjointe à une dichotomie entre occurrence et motif réutilisable. Les motifs centralisés sont tous réutilisables tandis que la plupart des motifs DIT ne le sont pas (ceux qui le sont constituent par ailleurs des cas particuliers).

La prédominance de la catégorie centralisée dans les motifs génériques et réutilisables s'explique surtout selon nous par le fait que ce style d'implémentation prévient les conflits section 5.2.2.5. Nous avons vu en effet qu'il est aussi possible d'utiliser les déclarations intertypes pour définir une solution réutilisable du motif Observer (section 5.2.1.3). À l'inverse, si les mécanismes et idiomes d'AspectJ ne permettent pas d'abstraire le motif pour le réutiliser, les déclarations intertypes sont utilisées pour supporter l'implémentation de l'occurrence.

Tableau 5.1 – Solutions des motifs du *GoF* avec AspectJ

Implémentation→ Abstraction ↓	Complément	Motif DIT	Motif aspectisé	Motif centralisé	Total
Motif générique	Prototype	Prototype	(Visitor)[a]	Composite *Flyweight* Command Singleton *Strategy*	6
Motif réutilisable	*Memento*[b]	CoR[c] Visitor	Proxy	CoR *Mediator* Observer	6
Occ.[d] extensible		Builder Bridge	Decorator*		3
Occ. spécifique		Adapter (class) Interpreter*	*State**		3
Orthogonal	Facade*[e] Iterator*	Factory Method Abstract Factory Template Method*			5
Total	3 [f]	9 [g]	3	8	23

[a] Nouvelles solutions section 5.3.2.
[b] Les motifs *accentués* offre seulement un support partiel.
[c] Motif Chain of Responsibility.
[d] Occurrence.
[e] Les motifs marqués d'une astérisque* n'utilisent pas d'interfaces-rôles.
[f] Prototype est compté uniquement en tant que motif DIT.
[g] Chain of Responsibility est compté uniquement en tant que motif centralisé.

Plus précisément, ceci s'explique par le fait qu'il est facile, avec AspectJ, d'abstraire et de réutiliser une structure et des collaborations entre interfaces-rôles à partir du moment où celles-ci sont génériques. Par contre il est difficile d'abstraire des méthodes dont la signature est spécifique au problème de l'application. C'est le cas par exemple des motifs Builder, Bridge et Adapter.

5.4.2.2 Prédominance des interfaces-rôles

La caractéristique la plus commune des solutions aspects est l'utilisation de l'idiome des interfaces-rôles. Seules 6 solutions sur les 23 ne les utilisent pas et font donc directement référence aux classes de l'application. Par ailleurs seuls certains rôles sont représentés par des interfaces. Cet idiome ouvre la voie à l'extension de ces rôles. C'est une condition nécessaire dans les motifs réutilisables et génériques.

5.4.2.3 Support de motifs

Le tableau 5.1 n'illustre pas complètement la contribution des aspects aux motifs. Pour plusieurs motifs, l'aide apportée par AspectJ est limitée et ne transforme qu'une partie de la

solution objet. Nous disons que l'implémentation ou l'usage du motif sont simplement supportés par les aspects. L'utilisation d'AspectJ implique toujours un travail d'implémentation de la solution objet.

C'est le cas des solutions classées orthogonales ou complémentaires. Ainsi la solution PROTOTYPE reste basée sur les mécanismes du clonage en Java. La solution ITERATOR encapsule la définition d'une classe `Iterator` et permet son accès via une occurrence de FACTORY METHOD. Les solutions FACTORY METHOD, ABSTRACT FACTORY et TEMPLATE METHOD, toutes basées sur la redéfinition de méthode, sont simplement extraites de l'arbre d'héritage simple avec des déclarations intertypes.

Certains motifs classés génériques ou réutilisables ne sont en fait supportés que partiellement. La partie fournie par l'aspect est réutilisable mais l'implémentation doit être complétée avec la solution objet. Ainsi la solution FLYWEIGHT implémente uniquement le gestionnaire des objets `Flyweight` et les opérations spécifiques aux problèmes. La solution STRATEGY implémente la relation entre les rôles `Context` et `Strategy` mais ne gère pas le changement de `Strategy`. La solution MEMENTO, classée réutilisable, déclare les interfaces-rôles mais ne définit aucune implémentation.

Les solutions MEDIATOR et STATE sont aussi partielles mais plus conséquentes. La solution MEDIATOR généralise la notification (à la façon du motif OBSERVER) mais pas le routage des notifications. La solution STATE propose de modulariser la transition entre états mais ne définit ni la relation entre les rôles `Context` et `State`, ni les états eux-mêmes.

La dispersion dans le tableau de cette notion de support reflète la disparité des motifs et des mécanismes associés, certains échappant à la programmation par aspects. Cependant la déclaration de signatures spécifiques à l'application – déjà remarqué en section 5.4.2.1 – semble être un problème commun à ces motifs supportés. Ce problème s'oppose à l'abstraction des solutions dans un aspect générique.

Usages particuliers Le projet de Hannemann *et al.* fait aussi la démonstration d'usages particuliers d'AspectJ, dans un but assimilé au support.

- La solution INTERPRETER montre l'utilisation de déclarations intertypes pour l'extension modulaire d'un ensemble de classes ; chaque traitement de la grammaire (interprétation, génération de code, vérification) est modularisé dans un aspect au lieu d'être dispersé dans chaque classe de la grammaire.
- La solution FACADE utilise le mécanisme post-compilation `declare warning` afin de vérifier la propriété principale du motif – c'est-à-dire l'absence d'appel, en dehors de la classe `Facade`, aux composants internes de l'occurrence.

5.4.2.4 Recoupement avec la classification originale des patrons

La classification originale des patrons du *GoF* (voir section 2.2.1.2 page 16) est basée sur deux critères. Ceux-ci donnent une partition très abstraite du but des patrons et de l'implémentation des motifs. Le premier critère définit trois finalités : création, structure, comportement. Ce critère indique le focus du problème visé par le patron. Le second critère définit la portée, classe ou objet. C'est un indicateur général sur la solution du motif dans un langage à classes. Une portée de classe implique uniquement des relations statiques entre classes, par héritage. Une portée objet implique en plus des relations dynamiques entre objets, qui sont définies par composition (agrégation ou association).

Les quatres motifs de portée classe (Adapter (class), Factory Method, Interpreter et Template Method) se retrouvent dans la catégorie des motifs DIT. Ceci est lié à l'usage des déclarations intertypes comme un mécanisme de réutilisation complémentaire ou transversal à l'héritage simple, comme l'héritage multiple, les mixins ou les traits (section 3.1.4.2 page 37).

La partition par le critère finalité est par contre complètement dispersée à travers le tableau. Ceci souligne le fait que les réponses apportées par les aspects ne reflètent pas forcément les problèmes visés par les patrons de conception de la programmation par objets.

5.4.3 Commentaires sur les cas particuliers

Nous commentons maintenant certaines solutions de Hannemann *et al.* présentant des particularités dans leurs propriétés ou leur utilisation des idiomes.

Proxy fait partie des motifs aspectisés. Ceci n'est pas surprenant étant donné sa proximité avec la solution objet Decorator – ce d'autant plus qu'un `Proxy` n'a pas besoin de la propriété récursive faisant défaut au motif Decorator aspectisé. Le motif Proxy émule plusieurs mécanismes dans un langage : la solution présentée par Hannemann *et al.* vise le contrôle d'accès à un objet. Elle peut être adaptée en `Proxy` virtuel pour l'instantiation paresseuse.

Par contre AspectJ ne permet pas d'implémenter un `Proxy` distant pour la programmation distribuée. Il manque dans ce cas l'association à une instance locale qui générerait les points de jonction spécifiques à l'application. Or créer une telle instance enlève l'intérêt de la transformation par les aspects.

Chain of Responsibility est la seule solution mixant déclarations intertypes et idiome de structure centrale. Si la chaîne des `Handler` est implémentée par la structure centrale, leurs réactions spécifiques sont définies par des déclarations intertypes. L'utilisation des DITs est nécessaire afin d'obtenir une chaîne récursive d'instances et adaptable par polymorphisme d'héritage.

Visitor a été présenté en détail en section 5.3.2. Il faut noter que la solution idiomatique de Visitor est le seul motif réutilisable basé uniquement sur les déclarations intertypes, bien que sa généricité soit très limitée. Nos nouvelles solutions, plus génériques, sont présentes dans la catégorie des motifs aspectisés.

Command définit deux relations avec l'idiome de structure centrale, de `Invoker` à `Command` et de `Command` à `Receiver`. Cette solution fait aussi la démonstration d'une interface mixte pour l'enregistrement de ces relations : avec une interface publique centralisée ou par des coupes-actions.

Adapter propose deux solutions (`class` ou `object`) en fonction des mécanismes du langage. Adapter `class` utilise l'héritage multiple et peut donc être émulé avec les déclarations intertypes. Par contre Hannemann *et al.* ne propose pas de solution pour Adapter `object`, qui utilise la composition d'objets. En effet le motif Adapter nécessite des signatures spécifiques à l'application, ce qui gêne la définition de collaborations génériques avec AspectJ.

State est un cas particulier où l'aspect gère les transitions d'état du motif. La transformation complète de ce motif est gênée par le problème des signatures spécifiques dans la délégation du rôle `Context` au rôle `State`.

Composite, Flyweight sont deux motifs centralisés n'utilisant pas de coupes-actions pour des raisons particulières. La solution FLYWEIGHT est partielle et ne prend en compte qu'une partie des collaborations. La solution COMPOSITE utilise le motif VISITOR pour définir les comportements. Ce cas est détaillé en section 6.3.

5.5 Bilan

Nous avons expliqué par l'exemple les deux approches pour l'implémentation des motifs de conception en programmation par aspects en vue de leur modularisation. Cette démarche révèle les idiomes AspectJ nécessaires pour définir une conception objet dans un aspect.

5.5.1 Approche idiomatique

L'approche idiomatique transforme partiellement la solution objet avec les mécanismes aspects et modularise les autres éléments du motif dans l'aspect : cette approche tend à reproduire dans l'aspect la solution objet initiale. En particulier nous montrons que les aspects utilisent un idiome de structure dès qu'ils définissent les relations de compositions entre rôles. Cet idiome est important car il détermine l'implémentation des autres éléments du motif.

D'une manière générale, les caractéristiques d'une transformation idiomatique d'une solution objet vers une solution aspect sont :

- la modularisation du motif dans son ensemble par un aspect singleton ;
- la déclaration des rôles par des interfaces Java internes à l'aspect (idiome des *interfaces-rôles*) ;
- la définition des relations de composition entre rôles avec un idiome de *structure* (structure centrale ou structure intertype) ;
- la transformation des collaborations par des coupes-actions et des méthodes ;
- la centralisation dans l'aspect des interfaces publiques du motif (idiome de *l'interface centrale* dans une occurrence du motif FACADE).

La revue des solutions proposées par Hannemann *et al.* pour les motifs du *GoF* indique qu'un tiers des solutions sont définies avec ces caractéristiques : elles utilisent en particulier un idiome de structure. La plupart des solutions utilisent l'idiome des interfaces-rôles et, dans une moindre mesure, une interface centrale.

Ces idiomes sont intéressants car ils permettent d'abstraire les rôles dans la solution (avec les interfaces-rôles) pour favoriser la réutilisation et d'éviter les conflits en centralisant l'interface.

5.5.2 Approche par aspects

L'approche par aspects capture le problème visé par le patron dans une nouvelle solution, différente de la solution objet initiale. Seuls trois motifs DECORATOR, VISITOR et PROXY (dans certaines variantes) bénéficient de cette approche. Elle se caractérise par la simplification du motif et, dans le cas de VISITOR, par une meilleure généricité.

Ces solutions ont aussi des propriétés différentes de leurs homologues objets en terme de comportement : l'utilisation et l'instantiation sont cachées par le modèle d'AspectJ ; la récursivité ne peut pas être exploitée. Des idiomes d'état (section 5.3.1.2) ou de portée (section 5.3.2.3) permettent d'émuler les propriétés dynamiques des solutions objets.

De fait, les solutions aspects sont simples tant que les collaborations utilisent des objets en association temporaire avec l'aspect (par exemple des objets capturés dans le contexte d'une coupe). ces idiomes sont quasi-nécessaires dès que l'aspect est associé de façon durable à des objets, c'est-à-dire hors du contexte d'une coupe.

5.5.3 Idiomes d'AspectJ

Nous considérons les idiomes des interfaces-rôles et de l'interface centrale comme propres à la transformation idiomatique des solutions objets.

Par contre les idiomes relatifs à la structure (sections 5.2.1.2 et 5.2.1.3), à l'état (section 5.3.1.2) ou à la portée (section 5.3.2.3) d'un aspect pointent vers une caractéristique du langage AspectJ. Ils concernent tous la liaison des aspects aux objets, qu'il s'agisse de définir une relation de composition entre objets, ou d'associer l'état ou la portée d'un aspect à un objet.

Ceci souligne la difficulté à utiliser les aspects d'AspectJ comme des objets normaux. En premier lieu le modèle d'instantiation-portée d'AspectJ est responsable de ces contraintes (section 3.1.4.1 page 36).

5.5.4 Conclusion

La démarche de modularisation avec AspectJ proposée par [Hannemann et Kiczales, 2002] et que nous présentons en détail est effective sur de nombreux motifs. Cependant notre revue (section 5.4) permet de nuancer ce succès en fonction du degré de réutilisation atteint par les solutions. La moitié des solutions sont considérées comme des motifs réutilisables voire génériques. L'autre moitié peut être modularisée comme occurrences extensibles ou spécifiques à l'application. Le résultat de la transformation est cependant très disparate : certains motifs ne sont que partiellement définis. D'une manière générale, les motifs non réutilisables bénéficient du support d'AspectJ pour les extensions structurelles à l'aide des déclarations intertypes.

La centralisation dans un aspect est en opposition avec la distribution des responsabilités entre participants objets (section 5.2.2.5). L'aspect change les moyens d'adaptation d'un motif. Le motif aspectisé Visitor fait la démonstration d'un polymorphisme ad hoc (section 5.3.2.4). Mais nous notons qu'il est actuellement impossible d'étendre une coupe ou une action avec AspectJ (section 5.2.2.3). À l'inverse les déclarations intertypes permettent l'adaptation par polymorphisme d'héritage. La solution Chain of Responsibility (page 96) illustre ce principe : si la structure est centralisée de façon idiomatique, les collaborations sont définies par des méthodes intertypes pour permettre la redéfinition.

Chapitre 6

Composition des motifs de conception avec AspectJ

Sommaire

Nous disons qu'il y a composition de motifs dès que les rôles de deux occurrences différentes se superposent sur une seule classe. Nous proposons le concept de modalité pour décrire différents niveaux de modularité dans la composition de motifs. Nous illustrons deux modalités différentes avec la composition des motifs Composite et Visitor ainsi que la réutilisation d'une telle composition.

Nous examinons les propriétés pour la composition des deux nouveaux mécanismes d'AspectJ : les déclarations intertypes et les coupes. Les déclarations intertypes proposent un mécanisme de composition structurelle similaire aux traits mais n'ont pas la finesse de ceux-ci pour la résolution des conflits. Le langage de coupe d'AspectJ permet la résolution d'interactions à partir du flot de contrôle.

6.1 Introduction

Comme indiqué en section 2.2.3, l'étude de la composition est restée limitée au niveau abstrait et informel d'une discussion des relations entre patrons. Ceci tient en particulier à deux facteurs :

- la disparité des motifs (section 2.2.1.5) gêne la modélisation générique de l'implémentation ;
- le manque de modularité des motifs (section 2.2.2 page 18) rend la compréhension d'une composition et de son impact plus difficile (chapitre 4).

Le chapitre 5 montre la modularisation de nombreux motifs par une transformation idiomatique avec AspectJ. Nous pouvons maintenant nous intéresser à la description des compositions de motifs. Nous disons qu'il y a composition de motifs dès que les rôles de deux occurrences différentes se superposent sur une seule classe.

Dans un premier temps nous décrivons avec le concept de *modalité** la composition des motifs comme s'il s'agissait d'entités de première classe. Ce concept nous fournit une grille de lecture conceptuelle sur les relations entre motifs dans le code. En particulier elle crée un lien avec l'implémentation en hiérarchisant la modularité des compositions. Cependant, si cette grille est applicable à tous les motifs, son analyse est restreinte à des considérations générales. Elle ne prend pas en compte les spécificités de chaque motif.

Dans un second temps il nous faut donc réduire chaque motif aux éléments qui le constituent et aux mécanismes qu'il utilise pour implémenter les compositions. Or la meilleure modularisation des motifs par les aspects rapproche les implémentations aspectisées d'entités de première classe. Cette approche nous permet donc d'appréhender les mécanismes concrets de composition entre motifs.

Nous examinons en détail les mécanismes de composition structurelle et comportementale d'AspectJ du point de vue de la résolution d'interactions et de conflits. La composition structurelle par les déclarations intertypes est similaire au modèle offert par les traits [Schärli *et al.*, 2003]. La finesse des opérateurs des traits pour la résolution des conflits fait cependant défaut aux déclarations intertypes. Le langage de coupe d'AspectJ constitue un nouveau mécanisme de composition comportementale. Nous montrons en particulier comment ce langage résoud par des coupes sur le flot de contrôle les interactions entre les motifs Observer, Composite et Decorator exposées en section 4.3.3.

6.2 Étude conceptuelle de la composition

Les relations de Zimmer (section 2.2.3.2 page 22) sont abstraites par rapport aux motifs eux-mêmes : elles ne donnent pas d'indication sur la composition et son implémentation. Nous considérons les motifs comme entités de première classe. Notre objectif est de donner une expression de la composition au niveau motif. Cette expression doit être applicable à tous les motifs et donner une indication sur l'implémentation. Plus précisément nous nous intéressons à la modularité de ces compositions.

Nous commençons donc par donner la définition minimale 6.1 pour la composition des motifs. Nous développons ensuite cette définition avec le concept de modalité, qui détaille la façon dont les motifs se composent.

> Définition 6.1 – Composition de motifs
>
> Il y a composition de motifs dès que les rôles de deux occurrences différentes se superposent sur la même classe.

6.2.1 Modalité d'une composition

Nous faisons l'hypothèse que toute composition de motifs n'est pas nécessairement modulaire. Si deux occurrences sont spécialisées pour un problème spécifique, ou si un motif en utilise un autre, il peut exister un compromis entre modularité et imbrication du code à des fins de simplification de l'implémentation. Dans les cas où les motifs interagissent au point de modifier leurs spécifications (section 4.3 page 61), la composition ne peut plus être modulaire.

Nous définissons le concept de *modalité** pour exprimer le compromis de modularité dans une composition de motifs.

> Définition 6.2 – Modalité d'une composition de motifs
>
> La modalité d'une composition de motifs exprime le degré d'imbrication (ou de dépendances) des deux motifs attendu dans la composition.

C'est une abstraction conceptuelle de l'impact de la composition sur l'implémentation. Une modalité correspond à une déclaration d'intention, par le programmeur, de la modularité *visée* dans l'implémentation des motifs. Après implémentation, elle permet d'évaluer si la modularité *atteinte* est adéquate. Nous décrivons cinq modalités :

Cohabitation : les deux motifs n'ont pas connaissance l'un de l'autre, ce qui n'empêche pas une proximité éventuelle (au sein de la même classe par exemple).

Collaboration : les deux motifs collaborent via leurs interfaces respectives.

Utilisation : un des motifs utilise l'autre et l'encapsule complètement.

Partage : deux motifs partagent une partie de leur implémentation (à fin de réutilisation par exemple).

Interaction : les deux motifs mis en présence impliquent une transformation pour implémenter la composition.

Des descriptions ci-dessus il apparaît que les trois premières modalités impliquent le respect de la modularité suivant une implication de plus en plus large des motifs : les dépendances sont absentes ou modulaires. Les deux dernières modalités sont non-modulaires. Dans le cas du partage, les spécifications des deux motifs sont fusionnées (section 4.3.4.1 page 69). Toute modification de la partie partagée doit donc satisfaire les deux motifs. Dans le cas d'une interaction, la composition induit une nouvelle spécification qui ne peut être implémentée que par une modification non modulaire d'un ou des deux motifs (section 4.3.3 page 65).

Une autre propriété de la modalité consiste en la symétrie ou non de la composition. Une composition asymétrique implique qu'un seul des deux motifs a connaissance de l'autre : c'est le cas de l'utilisation. Inversement, une composition symétrique implique que les deux motifs ont connaissance l'un de l'autre : c'est le cas du partage. Les modalités de collaboration et d'interaction peuvent être indifféremment symétriques ou asymétriques. Cette notion est invalide dans le cas de la cohabitation où la composition est « transparente ».

6.2.2 Impact d'une composition sur l'implémentation

Le corollaire de la modalité est l'impact, qui évalue de façon objective l'imbrication *réelle* des implémentations de deux motifs. Nous utilisons les quatres catégories définies par [Cacho *et al.*, 2006] :

Invocation : la composition implique l'appel de méthodes mais les implémentations ne concernent pas les mêmes classes.

Entrelacement intraclasse : les implémentations ciblent les mêmes classes sans mélanger leurs éléments respectifs.

Entrelacement intraméthode : les implémentations ciblent les mêmes méthodes, en conséquence de quoi leurs comportements sont mélangés dans ces méthodes.

Recouvrement : les implémentations partagent et utilisent les mêmes éléments. Il n'y a pas de mélange mais bien utilisation conjointe.

Même si certaines similitudes semblent évidentes entre ces catégories et les modalités, la divergence tient (outre le caractère déclaratif contre le caractère constatatif) dans le niveau d'abstraction : une modalité concerne la composition dans son ensemble tandis que l'impact s'applique aux éléments de la composition. Une composition peut donc cumuler plusieurs impacts mais déclare une seule modalité.

Tableau d'implication de la modalité vers l'impact

Le croisement de la modalité et de l'impact permet de juger de la modularité correcte d'une composition. Le tableau 6.1 définit les catégories d'impact correspondant aux modalités. Cet impact peut être symétrique ou non.

Tableau 6.1 – Implication des modalités vers les impacts. Un astérisque marque que la modalité (peut) implique(r) l'impact.

Impact→ Modalité ↓	Invocation	Entr. intraclasse	Entr. intraméthode	Recouvrement
Cohabitation		(*)		
Collaboration	*			
Utilisation	*	*		
Partage		*		*
Interaction			*	(*)

6.2.3 Mécanismes des compositions structurelles et comportementales

La composition conceptuelle des motifs par les modalités ne permet pas de prendre en compte les mécanismes effectifs du langage ni les spécificités de chaque motif. Nous classons ces mécanismes dans les deux domaines classiques des langages (section 3.1.4 page 36).

Le domaine structurel concerne la déclaration et la définition des éléments du langage : classe, interface, attributs et méthodes dans les langages à classes.

Le domaine comportemental concerne les collaborations et interactions entre ces éléments.

Les domaines discriminent les mécanismes par leur finalité. Dans les langages à classes, les mécanismes structurels pour la composition sont l'héritage et l'association. Les mécanismes comportementaux pour la composition sont l'appel polymorphique et l'accès aux attributs.

Les langages d'aspects, et AspectJ en particulier, offrent de nouveaux mécanismes pour la composition. Le mécanisme des coupes et actions permettent la composition comportementale. Les déclarations intertypes permettent la composition structurelle.

6.2.3.1 Réduction des modalités aux mécanismes

Un motif est implémenté avec des éléments des domaines structurel et comportemental. Par conséquent l'implémentation de la composition impliquera au moins l'un de ces domaines. C'est pourquoi nous examinons comment la composition conceptuelle, ou modalité, se réduit à une implémentation par les mécanismes.

- La cohabitation relève du domaine structurel, mettant en jeu les mécanismes des interfaces et de la réutilisation.
- À l'inverse la collaboration implique le domaine comportemental, par appel de méthode. Mais elle peut aussi demander l'implémentation structurelle d'une agrégation.
- Enfin l'utilisation relève des deux domaines : dans l'utilisation il ya à la fois cohabitation et collaboration.
- Le partage est similaire en principe à l'utilisation, bien qu'il révèle parfois un problème de réutilisation (section 4.3.4.1 page 69).

Nous verrons en section 6.3 les cas comparés de la collaboration et de l'utilisation dans la composition des motifs Composite et Visitor.

Le cas des interactions est plus complexe car l'impact et la résolution sont différents suivant le domaine.

- Une interaction structurelle peut se produire dans un graphe d'héritage où l'implémentation est indirecte (voir le cas section 4.3.4.2).
- Une interaction comportementale implique le contexte dynamique pour déterminer le comportement adéquat, tel qu'illustré en section 4.3.3.

Enfin nous classons aussi sous le terme d'interactions les conflits, c'est-à-dire les cas où le résultat de la composition présente une ambiguïté. Une interaction structurelle peut aboutir par exemple à un conflit de nom entre deux éléments. Une interaction comportementale peut aboutir à un conflit d'ordonnancement entre deux expressions.

Les sections suivantes présentent les nouveaux mécanismes de composition d'AspectJ en explorant différentes modalités (sections 6.3 et 6.5) mais aussi à travers le modèle des traits (section 6.4).

6.3 Étude de la composition de Composite et Visitor

L'intention du patron COMPOSITE, suivant Gamma *et al.*, est de « composer des objets dans une structure arborescente pour représenter une hiérarchie partie-tout. COMPOSITE permet aux clients de traiter uniformément objets individuels et composition d'objets. »

Le patron VISITOR, dont nous avons étudié l'aspectisation section 5.3.2 (page 87), propose de « représenter une opération à exécuter sur les éléments d'une structure d'objets. (...) »

La complémentarité de ces deux patrons est évidente et leur composition est bien connue, bien qu'ils puissent être utilisés indépendamment l'un de l'autre. Le but d'une telle composition est la séparation entre la description d'une structure arborescente à l'aide du motif COMPOSITE et la définition des opérations par le motif VISITOR.

La définition du motif VISITOR pose la question de l'itération sur la structure, sujet auquel le motif COMPOSITE est évidemment lié. Le *GoF* indique trois solutions possibles.

1. L'objet de la structure – en l'occurrence toute instance de `Composite` – prend en charge la traversée de ses propres composants. Cette traversée est donc récursive.
2. Un itérateur est défini pour parcourir toute la structure (suivant une stratégie particulière).
3. Le `Visitor` décide dynamiquement du prochain élément à traverser.

La première option est la plus courante car elle permet de préserver l'encapsulation de la structure. La deuxième est très proche de la première car c'est généralement la structure qui définit son propre itérateur : le parcours est cependant aplati par rapport au parcours récursif précédent. La dernière option est réservée à des cas particuliers et car elle implique une connaissance intime par le `Visitor` de la représentation de la structure.

Nous utilisons la première option d'itération : la classe `Composite` définit son propre itérateur récursif pour appeler `Visitor`. Ce dernier n'a donc pas besoin de détails sur la représentation interne à `Composite`.

Nous considérons deux modalités pour étudier la composition. La collaboration suppose que le motif VISITOR est défini hors du cadre du motif COMPOSITE. Ceci permet l'emploi d'une solution VISITOR générique (section 5.3.2.2), ou encore, si l'interface de `Composite` le permet, l'emploi d'un itérateur externe par `Visitor`. L'utilisation implique au contraire que le motif VISITOR est encapsulé dans COMPOSITE et n'est pas utilisable hors de ce contexte.

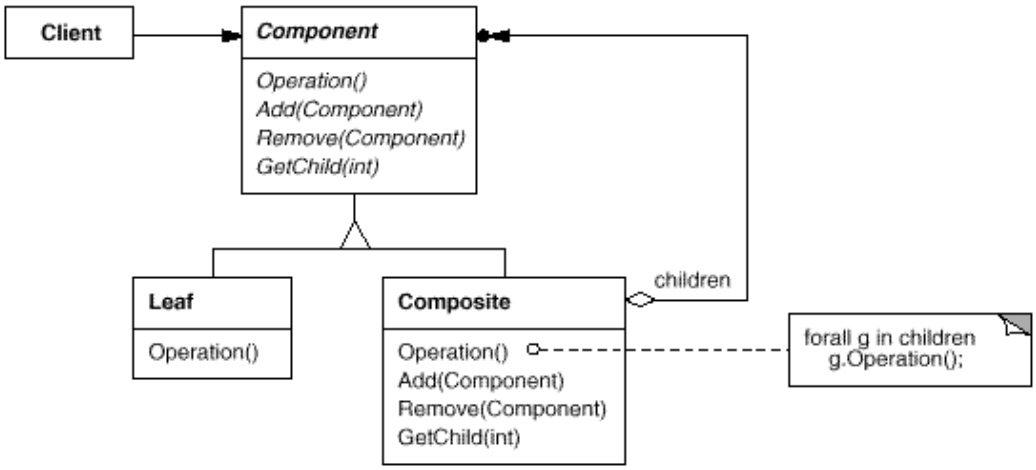
6.3.1 Description de la solution objet pour Composite

La solution est basée sur une structure de classe récursive (figure 6.1) : la classe `Composite` tout à la fois hérite et définit une relation un-à-plusieurs à la classe de base `Component`, qui représente n'importe quel objet de l'arbre. Une instance de `Composite` peut ainsi référencer toute instance du type `Component`, inclus le `Composite` lui-même.

La classe `Composite` hérite de l'interface de `Component` mais redéfinit en principe chaque méthode afin de, au minimum, la rediriger vers ses composants. Par récursion, une méthode appelée sur un `Composite` descend la chaîne des composants jusqu'aux instances non `Composite`.

Problèmes liés à la réutilisation et à l'implémentation. Le motif COMPOSITE présente a priori peu de caractéristiques propres à l'aspectisation. Son implémentation est encapsulée dans la seule classe dédiée `Composite`. Cependant deux caractéristiques du langage Java gênent la définition d'un motif COMPOSITE réutilisable.

Illustration 6.1 – Diagramme de classes illustrant la structure du motif COMPOSITE (extrait du *GoF*). Le code de `Operation()` décrit le comportement récursif attendu.



Premièrement la classe `Composite` doit être un sous-type de `Component` pour permettre l'homogénéité et la récursivité de la structure. Ceci passe généralement par l'héritage de la classe `Component`. Du fait de l'héritage simple en Java, il n'est pas possible d'avoir une classe `Composite` héritant d'un côté de `Component` et de l'autre d'une implémentation générique du motif COMPOSITE.

Deuxièmement, faute de multi-méthodes, *closures* ou fonctions de première classe, la définition des opérations récursives dans la classe `Composite` mène à dupliquer le code d'itération pour chaque méthode. Il y a donc un surcoût à l'implémentation du motif.

Or la composition de COMPOSITE et VISITOR est une solution, quoique lourde, à ce dernier problème. En réifiant l'opération dans un objet `Visitor`, il est possible de passer celui-ci en paramètre à un unique itérateur générique.

6.3.2 Collaboration des motifs Composite et Visitor

Le but de l'aspectisation et de la composition des motifs COMPOSITE et VISITOR est donc d'obtenir une solution réutilisable pour COMPOSITE. Le motif COMPOSITE est implémenté séparément puis appliquer aux classes visées, indépendamment de leur hiérarchie. Notre solution pour VISITOR, montrée section 5.3.2.2, est suffisamment générique pour être réutilisée.

6.3.2.1 Définition de la composition réutilisable avec AspectJ

La solution montrée (code source 6.1) utilise l'idiome de structure centrale, bien qu'une solution utilisant les déclarations intertypes soit aussi possible. L'aspect `CompositeProtocol` encapsule toutes les définitions. Les interfaces-rôles sont définies (lignes 3 à 5). La table centrale `perComponentChildren` garde pour chaque instance de `Composite` un `Vector` de `Component` (ligne 8). Cette solution est inspirée, pour la partie structurelle, de la solution de Hannemann *et al.*.

Code source 6.1 – Motif COMPOSITE réutilisable en collaboration avec le motif VISITOR générique.

```
1  public abstract aspect CompositeProtocol {
2   // Roles
3   public interface Component {}
4   public interface Composite extends Component {}
5   public interface Leaf      extends Component {}
6
7   // Structure centrale
8   private Map perComponentChildren = new WeakHashMap();
9   private Vector getChildren(Component s) {
10   Vector children = (Vector)perComponentChildren.get(s);
11   (...) // instantiation paresseuse si necessaire
12   return children; }
13
14  // Interface d'enregistrement
15  public void addChild(Composite cs, Component cn) {
16   getChildren(cs).add(cn); }
17  public void removeChild(Composite cs, Component cn) {...}
18  public Iterator traverseChildren(Component c) {
19   (...) return getChildren(c).iterator(); }
20
21  // Collaboration avec Visitor
22  declare parents: Component extends VisitableNode;
23  public void recurseAll(Component c, Visitor visitor){
24   visitor.visit((VisitableNode) c);
25   for (Iterator ite = traverseChildren(c); ite.hasNext();)
26    recurseAll((Component) ite.next(), visitor); }
27 }
```

La composition des motifs COMPOSITE et VISITOR est définie lignes 22 à 26. Il s'agit ici d'une collaboration asymétrique entre deux motifs génériques : le motif COMPOSITE utilise l'interface de VISITOR. En particulier il n'est pas nécessaire de détailler les différents types de `Component` qu'un `Visitor` peut rencontrer. Seule la relation de `Component` à `VisitableNode` est déclarée (ligne 22). Il est alors possible de définir des itérateurs génériques tels que `recurseAll` (lignes 23 à 26). Chaque `Component` est visité par l'appel de la méthode `visit` (ligne 24) et parcouru récursivement (lignes 25–26).

6.3.2.2 Spécialisation et usage

Le code source 6.2 montre la spécialisation du motif COMPOSITE dans une occurrence et son utilisation avec VISITOR. La hiérarchie de classes `TestClass`, `TestLeaf` et `TestGroup` sert d'exemple. L'aspect `TestManager` étend `CompositeProtocol` (ligne 7) afin de définir une occurrence de Composite. À chaque classe de la hiérarchie est assignée une interface-rôle (lignes 8 à 10). Cette étape est la seule nécessaire pour utiliser `CompositeProtocol`.

Les lignes 13–14 montrent les tâches typiques d'instantiation et d'enregistrement. Nous utilisons ensuite une occurrence de VISITOR définie pour une structure générique (voir code source

Code source 6.2 – Spécialisation et usage de la collaboration COMPOSITE-VISITOR

```
1  // Classes de test
2  class TestClass {}
3  class TestLeaf extends TestClass  {}
4  class TestGroup extends TestClass {}
5
6  // Occurrence de Composite
7  aspect TestManager extends CompositeProtocol {
8   declare parents: TestClass extends Component;
9   declare parents: TestLeaf extends Leaf;
10  declare parents: TestGroup extends Composite; }
11
12 // Usage de Composite et Visitor
13  TestGroup root = new TestGroup();
14  TestManager.aspectOf().addChild(root, new TestLeaf());
15  CountingVisitor visitor = new CountingVisitor();
16  TestManager.aspectOf().recurseAll(root, visitor);
17  System.out.print("Nombre de feuilles : " + visitor.report());
```

5.11, page 90) pour compter le nombre de nœuds dans la structure. Un `Visitor` spécifique à cet exemple sera défini et utilisé de la même manière.

6.3.3 Utilisation de Visitor par Composite

[Hannemann et Kiczales, 2002] propose pour le motif COMPOSITE la solution présentée dans le code source 6.3. Si la structure ne change pas, il est intéressant de comparer la partie comportement (lignes 10 à 14) qui, comme pour notre solution, définit des itérateurs génériques grâce au motif VISITOR.

L'examen de cette implémentation montre que l'occurrence de VISITOR est dégénérée. Celle-ci définit une interface `Visitor` très simple avec la seule méthode `doOperation` (lignes 10–11). Cette définition de VISITOR est plus proche du motif COMMAND que d'un mécanisme de double envoi. Cependant cette interface est suffisante pour être utilisée par `Composite` dans la définition de la fonction d'itération générique `recurseOperation` (lignes 12–14). Il faut noter d'ailleurs que cette fonction d'itération n'est pas récursive, contrairement à ce que son nom laisse supposer.

Le code source 6.4 montre la spécialisation et la définition d'une opération avec `Visitor`. L'opération est définie par une classe interne `CountVisitor` (lignes 13 à 17) et les méthodes intertypes `count` introduites dans les classes de la structure (lignes 7 à 11). Une récursion croisée est nécessaire entre `count` appellant `recurseOperation` (ligne 10) et `doOperation` appellant `count` (ligne 16). C'est donc au programmeur de boucler la récursion en faisant cet appel croisé. Ceci rend la compréhension modulaire de cette solution COMPOSITE moins aisée, puisque le flot de contrôle saute entre l'itérateur générique et la solution VISITOR spécialisée.

La modalité est ici clairement l'utilisation de VISITOR (ou COMMAND) par COMPOSITE. La solution VISITOR est complètement intégrée à l'implémentation de COMPOSITE et ne peut être exploitée indépendamment.

Code source 6.3 – Motif COMPOSITE réutilisable de Hannemann *et al.*, utilisant une occurrence VISITOR ad hoc. La partie structurelle est donnée dans le code source 6.1.

```
1  public abstract aspect CompositeProtocol {
2   // Roles
3   (...)
4   // Structure centrale
5   (...)
6   // Interface d'enregistrement
7   (...)
8
9   // Utilisation de Visitor
10  protected interface Visitor {
11   public void doOperation(Component c); }
12  public void recurseOperation(Component c, Visitor v) {
13   for (Iterator ite = traverseChildren(c); ite.hasNext();)
14    v.doOperation((Component) ite.next()); }
15 }
```

La nécessité d'intégrer cette solution VISITOR dédiée au motif réutilisable COMPOSITE est aussi liée, dans ce cas, à la non-généricité de la solution idiomatique VISITOR (section 5.3.2.1). L'extension de la solution COMPOSITE serait en effet gênée par la spécialisation limitée du `Visitor`.

6.3.4 Bilan

Cette étude montre qu'il est possible de définir une composition réutilisable des motifs COMPOSITE et VISITOR. La définition de la composition est illustrée avec deux modalités différentes. La collaboration est basée sur le motif générique VISITOR. L'utilisation définit une solution VISITOR dédiée au motif encapsulateur. Le résultat de la composition est sensiblement identique dans les deux cas, fournissant un motif COMPOSITE réutilisable et doté d'un itérateur générique.

La différence entre les deux compositions s'exprime par des propriétés de modularité, de simplicité d'implémentation et d'usage divergentes entre les deux solutions VISITOR.

La solution générique VISITOR est caractérisée par :

- la modularité entre COMPOSITE et VISITOR – la dépendance est à sens unique ;
- la définition par une forme standard des classes `Visitor`.
- la possibilité d'utiliser des occurrences elles-mêmes génériques (comme `CountingVisitor`) ;

La seconde solution (par récursion croisée) est caractérisée par :

- la forte intégration entre le motif abstrait `CompositeProtocol` et l'occurrence `CountVisitor` ;
- l'utilisation d'un idiome de récursion croisée spécifique aux classes `Visitor` ;
- une utilisation conjointe avec des méthodes intertypes pour définir les réactions polymorphiques.

Nous constatons que si la modalité d'utilisation pour la composition de motifs produit un résultat dédié à un problème particulier, la modalité de collaboration peut aboutir à un résultat modulaire, plus souple et intéressant. Cependant nous pensons que ce résultat n'est possible qu'avec des solutions suffisamment génériques.

Code source 6.4 – Spécialisation et usage de l'utilisation COMPOSITE-VISITOR.

```
1  // Occurrence de Composite et Visitor dedie
2  aspect TestManager extends CompositeProtocol {
3   declare parents: TestClass extends Component;
4   declare parents: TestLeaf extends Leaf;
5   declare parents: TestGroup extends Composite;
6
7   public int Component.count(){ return 1; }
8   public int Composite.count(){
9    CountVisitor v = new CountVisitor();
10   TestManager.aspectOf().recurseOperation(this, v);
11   return v.acc; }
12
13  public class CountVisitor implements Visitor {
14   int acc = 0;
15   public void doOperation(Component c) {
16    if( c instanceof TestLeaf ) acc += 1;
17    c.count(); } // appel recursif croise
18  }
19 }
```

6.4 Composition structurelle avec les déclarations intertypes

Les déclarations intertypes d'AspectJ apportent de nouvelles solutions pour la composition structurelle. Elles sont utilisées dans les implémentations des motifs pour supporter le polymorphisme d'héritage, comme dans le cas du motif CHAIN OF RESPONSIBILITY (section 5.4.3). Même si la transformation idiomatique d'un motif n'est pas intéressante (section 5.2.2), les déclarations intertypes peuvent supporter les motifs en facilitant leur usage ou capturant une partie de leur implémentation (section 5.4.2).

Or les déclarations intertypes partagent des similarités avec les traits de [Schärli *et al.*, 2003]. Les traits sont un mécanisme simple pour la réutilisation complémentaire de l'héritage simple (section 3.1.4.2 et encart page 39). Les deux modèles autorisent une composition « plate », c'est-à-dire sans encapsulation des éléments composés. Mais les traits proposent aussi des opérateurs à grain fin pour la résolution des conflits dans les compositions.

Cette section explore donc l'émulation des traits en Java avec AspectJ afin de comparer en détail les deux modèles. Nous exposons ainsi les convergences et les divergences des déclarations intertypes et des traits et en tirons les leçons pour l'amélioration d'AspectJ.

6.4.1 Définition d'un trait avec AspectJ

Le code source 6.1 est un exemple de trait défini avec les déclarations intertypes : `ColorAdapter` est une occurrence ADAPTER destinée à coloriser des figures en leur fournissant l'interface et les méthodes. À partir de cet exemple, nous expliquons progressivement les capacités des déclarations intertypes pour émuler les traits.

Exemple 6.1 – Émulation d'un trait avec les déclarations intertypes et les interfaces Java

```
1  interface ColorAdapter {  // interface trait
2    public int getRed();
3    public float getHue();
4    public boolean equals(Object o);
5    public Color getRGB(); // methode requise (non definie)
6  }
7  aspect ColorRepository {  // definition des methodes reutilisables
8    public float ColorAdapter.getHue() {
9      float[] hsb = Color.RGBtoHSB(getRed(),getGreen(),getBlue(),null);
10     return hsb[0]; }
11   public boolean ColorAdapter.equals(Object o) {
12     ... } // test egalite couleur avec o
13 }
14 // Usage
15 class Circle implements ColorAdapter {
16   private Color color;
17   public Color getRGB(){ return color; }
18 }
```

Une déclaration intertype permet la définition d'une méthode dans une classe hors du code source de celle-ci (exemple 3.2 page 37). Après compilation, la méthode fait partie intégrante de l'interface de la classe et se comporte comme si elle était directement définie dans la classe. Cette propriété est aussi celle de la composition plate des traits.

Cependant, pour arriver aux fins de réutilisation des traits, il est aussi possible de faire une déclaration intertype *en combinaison avec une interface Java* (exemple 6.1, lignes 8 et 11). Ceci ne change pas la nature des interfaces Java : elles sont simplement utilisées comme indicateurs par le tisseur. Toute classe déclarant une interface sera augmentée avec les éléments intertypes associés à l'interface. La réutilisation suivant le modèle des traits est acquise grâce à la transversalité par rapport à l'arbre d'héritage :

1. la déclaration intertype est liée à une interface Java, type abstrait dans la hiérarchie des classes ;
2. la déclaration `implements` d'une classe en Java est orthogonale à la déclaration d'héritage elle-même.

Par ailleurs cet usage des interfaces Java permet la déclaration des méthodes requises (ligne 5). Une méthode déclarée par l'interface mais non définie par l'aspect reste une contrainte d'implémentation pour la classe.

Il faut distinguer nettement, dans cette émulation des traits, les rôles de l'interface Java (`ColorAdapter`, lignes 1–6) et de l'aspect (`ColorRepository`, lignes 7–13). Les classes déclarent utiliser *l'interface* (et doivent implémenter en conséquence ses méthodes requises – lignes 15 à 18) pour bénéficier du trait : l'interface est l'élément public du trait. À l'inverse l'aspect fournit les implémentations mais est transparent pour les classes. Cette distinction se reflète dans notre convention de nommage : l'interface porte le nom du trait tandis que l'aspect porte le suffixe `Repository`.

6.4.2 Composition et résolution des conflits

La composition plate dans une classe est la propriété basique des traits. La définition d'une méthode par une déclaration intertype émule cette propriété. Nous allons maintenant examiner comment les cas de conflits peuvent être traités dans les classes et les traits composites définis avec les déclarations intertypes d'AspectJ.

6.4.2.1 Composition d'un trait dans une classe

La composition se fait avec la déclaration `implements` de l'interface Java du trait et l'implémentation des éventuelles méthodes requises.

La définition directe par une classe d'une méthode inhibe l'insertion par le tisseur d'une définition de même nom par une déclaration intertype. Cette priorité est la même que la composition par redéfinition des traits Smalltalk.

Le cas diffère si la méthode en question est héritée par la classe. AspectJ considère la déclaration intertype en conflit avec la méthode héritée et produit une erreur de compilation. Ce conflit est ignoré dans le modèle des traits Smalltalk. L'insertion de la méthode du trait est effectuée : par effet de la composition plate, celle-ci redéfinit la méthode héritée.

6.4.2.2 Composition de plusieurs traits

La composition de plusieurs traits dans une seule entité, classe ou trait composite, est possible grâce aux interfaces. En effet leur usage n'est pas contraint par la règle d'héritage simple. Une classe peut déclarer plusieurs interfaces et ainsi utiliser plusieurs traits. De même une interface peut étendre d'autres interfaces de façon à produire l'interface du trait composite.

Cependant cette composition de plusieurs interfaces dans une interface composite ne produit pas un trait composite à part entière. La composition est « retardée ». En effet le tisseur ne compose pas les implémentations à ce niveau. Elles seront introduites uniquement dans les classes utilisant le trait composite. Cette différence entre interface et trait a des répercussions sur les conflits dans les traits composites, comme montré dans la section suivante.

6.4.2.3 Conflits dans la composition des traits

La composition de plusieurs interfaces, comme de plusieurs traits, n'est pas hiérarchisée : elle introduit donc la possibilité de conflits entre des définitions de même nom. L'exemple 6.2 montre un cas de conflit dans la classe `Circle` quand elle est composée (ligne 12) avec deux Adapter, `ColorAdapter` et `AreaAdapter`. Les deux traits définissent chacun une méthode `equals(Object)`.

AspectJ gère ces conflits en les détectant et déclarant d'une erreur au tissage. En l'absence des opérateurs à grain fin que sont l'alias et l'exclusion, la résolution se limite à redéfinir la méthode dans la classe pour outrepasser le conflit (ligne 17 en commentaire), suivant la règle de priorité (section 6.4.2.1). La réutilisation d'une méthode de trait en conflit est cependant impossible par cette méthode.

L'artefact de définition des traits composites, via les interfaces composites, implique que les implémentations ne sont pas réellement composées avant l'utilisation par une classe. Ceci reporte la détection des conflits dans les traits composites au moment de leur utilisation par une classe, ce qui peut entraîner une cascade de conflits dans le cas d'une hiérarchie de traits.

Exemple 6.2 – Conflit entre deux traits définis avec AspectJ

```
1  interface AreaAdapter {
2    public int area();
3    public boolean equals(Object o);
4    public int compareTo(Object o);
5  }
6  aspect AreaRepository {
7    public boolean ColorAdapter.equals(Object o) {
8      ... } // test egalite superficie avec o
9  }
10
11 // Conflit ColorAdapter.equals et AreaAdapter.equals
12 class Circle implements ColorAdapter, AreaAdapter {
13   private Color color;
14   public Color getRGB(){ return color; }
15   public int area() { ... }
16 // resolution du conflit par redefinition
17 //  public boolean equals(Object o) { ... }
18 }
```

Cependant une méthode dans un trait composite a priorité sur une méthode d'un trait utilisé (par le composite). Cette règle est aussi valable avec AspectJ. Ceci permet de prévenir les conflits en redéfinissant une déclaration intertype dans l'aspect. Le traitement des conflits dans les traits composites avec AspectJ est donc similaire à celui des classes mais aussi handicapé par l'absence de détection précoce.

Le traitement des conflits avec AspectJ dans les classes comme dans les traits composites est donc limité à la redéfinition des méthodes en conflit. Ce principe ne permet pas la réutilisation des méthodes cachées.

Pour parvenir à une réutilisation complète, il faut passer par la délégation d'objet. Au lieu d'utiliser directement plusieurs traits, un objet peut déléguer à chacun de ses objets l'utilisation d'un seul trait. L'encapsulation est rétablie, ce qui élimine les conflits au prix d'un surcoût à l'implémentation et à l'exécution. Cependant cet idiome fait perdre une grande part de l'intérêt des traits.

6.4.2.4 Déclaration intertype d'état et conflit

Le modèle des traits n'autorise pas la définition d'un état dans un trait afin d'éviter l'ambiguïté liée à l'héritage diamant. À l'inverse les déclarations intertypes en AspectJ permettent d'introduire un champ dans une classe ou via une interface.

Cette possibilité est encadrée principalement par deux caractéristiques. Premièrement, les déclarations intertypes divergent de l'héritage multiple puisque l'usage associé à ce cas de figure, c'est-à-dire l'implémentation de plusieurs interfaces, entraîne une composition « retardée » (section 6.4.2.2). La définition d'un champ dans une relation diamant entre des interfaces et une classe se réduit alors à la définition du champ dans la classe. Deuxièmement, le contrôle d'accès est possible avec les modifieurs `public`, `private` ou défaut. En particulier, le modifieur `private` restreint l'accès au seul aspect faisant la déclaration. Ce modifieur empêche donc la réutilisation.

La déclaration publique d'un champ *via* une interface reste risquée car :
- cela rend l'état accessible à tout objet ;
- un conflit est toujours possible avec une autre déclaration.

Il est donc recommandé de déclarer les champs en `private` afin d'obtenir l'encapsulation au détriment de la réutilisation.

6.4.3 Bilan

Les déclarations intertypes offrent une émulation basique des traits en Java. Ceci est lié à la composition « plate », où les méthodes sont directement introduites dans la classe, et à la transversalité par rapport à l'héritage classique.

Cependant il existe plusieurs divergences dans la gestion des conflits : certaines priorités sont différentes et AspectJ propose les modifieurs de Java pour encapsuler les déclarations au détriment de la réutilisation. Le modèle des traits propose des opérateurs à grain fin sur les signatures permettant de configurer chaque composition. Grâce à cette précision, ce modèle se révèle plus facile à utiliser que les déclarations intertypes.

6.4.3.1 Perspective : impact sur le typage statique

Dans le cadre d'une implémentation complète du modèle des traits en Java et AspectJ, comprenant donc les opérateurs de résolution des conflits, il est intéressant de considérer l'impact du typage statique sur les traits. Nous discutons deux points relatifs au typage : la question du trait en tant que type et les risques de conflit lié au type retour des méthodes requises.

La question du typage statique des traits peut être considérer suivant deux options opposées : la première considère un trait comme un type nominatif, intégré au système normal des classes et des interfaces Java ; la deuxième considère un trait comme une entité avec un type paramétrique, orthogonal aux types nominatifs.

Trait comme type nominatif. Le trait `T` définit un type T associé, à la manière d'une classe ou d'une interface. Le trait type ses propres instances (dans les méthodes binaires – `T.f(T t)` – par exemple) en utilisant le type T. Ceci permet une grande généricité, en particulier le polymorphisme entre classes utilisant le même trait.

Cependant les opérateurs de résolution de conflit perturbent la relation de sous-typage entre une classe et un trait. En particulier, l'opérateur d'exclusion casse cette relation. Il existe deux solutions : soit changer la sémantique de l'exclusion pour une redéfinition forcée – dans ce cas l'interface du trait reste garantie ; soit considérer que l'exclusion définit effectivement un nouveau type anonyme, « assimilé » à la classe utilisatrice et indépendant du type trait original. L'alias peut dans ce cas aussi définir un type assimilé à la classe, tout en conservant une relation de sous-typage avec le trait.

Cette démarche est la plus classique dans le cadre de Java et d'AspectJ, lequel fait déjà usage des interfaces comme intermédiaire générique. Elle implique cependant la prise en compte des opérateurs dans le système de type.

Trait avec type paramétrique. Le trait `T` définit un type paramétrique T. Chaque classe substitue son type à T quand elle utilise le trait. La relation au trait original n'existe donc plus. Par exemple les méthodes binaires sont spécialisées pour la classe utilisatrice : elles permettent

directement le polymorphisme avec les instances de la classe mais pas avec d'autres classes utilisant le trait. Cette solution, a priori moins générique que la précédente, met l'accent sur la réutilisation et le paramétrage plutôt que sur le sous-typage. Le sous-typage pour polymorphisme peut toujouts être appliqué avec les interfaces Java.

Si les génériques en Java permettent l'utilisation de types paramétriques dans une entité, elles ne permettent pas, à l'heure actuelle, la déclaration de l'entité elle-même comme type paramétrique (type $Self$). Sans ce type $Self$, cette solution perd l'avantage de la simplicité.

Conflit du type retour dans les méthodes requises. L'introduction d'un typage statique peut réduire le nombre de conflits en discriminant mieux les signatures des méthodes. Cependant le système de types de Java expose au piège suivant. Le type retour d'une méthode concrète n'est pas pris en compte dans sa signature. Deux méthodes dont seul le type retour diffère sont donc en conflit. Par contre l'implémentation d'une méthode abstraite doit respecter ce type retour. Le programmeur peut donc être bloqué dans un cas pathologique : il est impossible de satisfaire deux méthodes requises dont seul le type retour diffère, car leurs implémentations sont alors en conflit.

6.5 Interaction comportementale par les coupes

Une interaction se produit entre deux motifs quand au moins l'un d'entre d'eux nécessite une adaptation pour s'accomoder du contexte modifié par l'autre. Cette adaptation peut se traduire par des intrusions de code et, dans certains cas, par des conflits. Les aspects permettent de limiter les intrusions par des coupes (l'interaction persiste néanmoins si la coupe doit être modifiée). Nous avons vu dans le domaine structurel le cas des conflits de nom, que le modèle des traits résoud de façon élégante (section 6.4). Dans le domaine comportemental, un conflit se traduit par un problème d'ordonnancement entre les expressions du programme : AspectJ propose la notion de précédence pour résoudre un tel problème.

Exemple : interactions dans le flot de contrôle. Cependant une interaction ne se résume pas forcément à une intrusion ponctuelle. Nous explorons dans cette section les interactions à l'échelle de plusieurs objets en relation et leur adaptation dans le contexte du flot de contrôle. Comme support à cette réflexion, nous nous intéressons aux interactions entre les motifs Observer, Composite et Decorator exposées en section 4.3.3 (page 65).

Les motifs Composite et Decorator définissent par nature une structure d'objets récursives. Nous appelons *structure d'objets* tout ensemble d'instances pour lequel il existe un ordre partiel, défini par exemple par un parcours hiérarchique de la structure.

Le motif Observer est l'indication d'une dépendance abstraite d'un observateur à un sujet (sections 2.2.2.2 et 3.2.2). Or l'introduction des motifs Composite et Decorator sur le sujet induit un changement d'échelle : le sujet devient une structure. La dépendance de l'observateur doit être adaptée à ce nouveau contexte. Par la suite nous appelons *stratégie* une adaptation de l'observateur à une interaction particulière avec la structure.

6.5.1 Stratégies d'observation dans une structure d'objets

De façon plus concrète, une stratégie d'observation définit la condition de notification de l'observateur. Cette condition prend en compte deux paramètres : une action et son contexte

d'exécution. Précisément ce contexte est constitué par le parcours effectué dans la structure d'objets, comme montré en section 4.3.3 (page 65).

Une autre caractéristique importante est la capture du contexte afin de paramétrer la notification. Ce contexte, bien que son utilisation dépende de la fonction de l'observateur, peut être représenté par un unique objet. Il peut suffir de passer la référence à l'objet courant, désigné alors comme « responsable » de la notification. Nous avons cependant vu comment le motif DECORATOR prend la main sur l'objet courant en tant que responsable (section 4.3.3.2 page 66).

En résumé nous avons trois paramètres essentiels pour définir une stratégie :

- l'action ciblée ;
- le parcours déclenchant la notification, fonction de la structure ;
- le contexte paramétrant la notification.

Dans notre approche la structure et le contexte se distinguent comme des paramètres formalisés de la notification, au même titre que l'action.

À cela il faut rajouter une caractéristique lors de la définition conjointe de plusieurs actions : le grain (ou portée) de la stratégie définit la précision avec laquelle celle-ci peut cibler une ou un ensemble d'actions. Le grain peut varier par exemple de l'échelle d'une classe (la stratégie s'applique à toutes les actions de la classe) à l'échelle d'une méthode (dans ce cas, plusieurs stratégies peuvent cohabiter au sein d'une classe).

6.5.1.1 Description des stratégies d'observation

Nous proposons quatre stratégies fondées sur un parcours hiérarchisé (de haut en bas, par exemple récursif) d'une structure (figure 6.2). Bien que ces quatre cas ne soient pas exhaustifs, ils représentent à notre avis les plus courants dans le cadre d'un parcours classique d'une structure d'objets.

Nœuds **:** tous les nœuds de la structure déclenchent une notification.

Feuilles **:** seules les feuilles de la structure déclenchent une notification.

Sommet **:** une notification est déclenchée au sommet du parcours hiérarchique d'une action.

Branche **:** la notification déclenchée par un nœud est subsumée par un nœud parent.

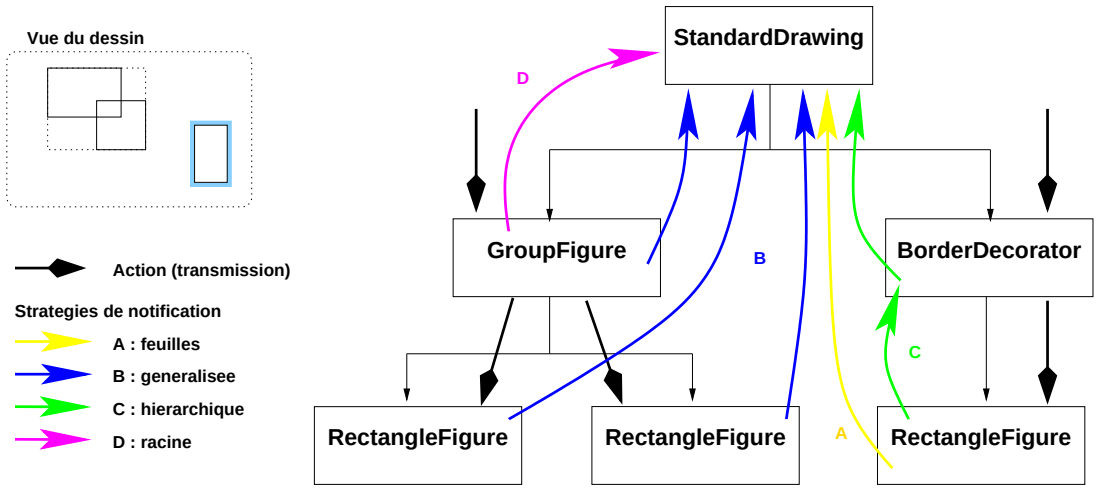
La stratégie *nœuds* est standard et ignore de fait l'impact de la structure sur la notification : cette stratégie est sans risque si des notifications redondantes n'ont pas d'effet indésirable. Par opposition les stratégies *feuilles* et *sommet* cherchent à réduire la redondance des notifications, mais suivant des hypothèses différentes. Le cas des feuilles est utile dans l'hypothèse où seules celles-ci peuvent décider de la pertinence d'une notification – les parents sont alors incompétents pour en juger. Le cas des sommets juge au contraire le parent compétent et évite la redondance dans les feuilles. La stratégie *branche* est la plus particulière puisqu'elle implique l'existence d'une dépendance entre deux nœuds dans la branche pour déclencher une notification : il s'agit pour un nœud parent de prendre en charge une notification initialement provoquée par un nœud fils. Ce cas s'applique entre autre au motif DECORATOR.

La capture du contexte de la notification reflète les stratégies hiérarchiques ci-dessus : le contexte est pris dans la branche courante du parcours.

Nœud courant : c'est le paramètre standard d'une notification.

Nœud parent : un parent est désigné comme responsable de la notification, généralisant le contexte de celle-ci.

Illustration 6.2 – Stratégies de parcours dans une structure d'objets, suivant l'exemple de la section 4.3.3.2.



Nœud fils : un fils est désigné comme responsable, spécialisant le contexte.

Malgré cette similarité, la capture de contexte doit pouvoir être combinée avec n'importe quelle stratégie de parcours. Cette possibilité permet d'émuler un comportement en l'absence de la stratégie adéquate, comme montré dans le cas d'AspectJ en section suivante.

Nous avons vu en section 4.3.3.3 (page 67) l'implémentation de certaines de ces stratégies dans un langage à classes. En particulier la stratégie *branche*, bien que subtile, peut être émulée grâce au motif Chain of Responsibility. Cependant le coût d'implémentation est conséquent et conduit à utiliser la même stratégie pour toutes les actions de la classe. Le grain est donc en général à l'échelle de la classe.

6.5.1.2 Définition des stratégies avec un langage de coupe

Nous posons l'hypothèse suivante : un parcours hiérarchisé de la structure définit un flot de contrôle hiérarchisé. La définition des stratégies peut se faire grâce à un langage de coupe sur le flot de contrôle. Nous nous inspirons en particulier du modèle présenté en [Douence et Teboul, 2004], tout en l'adaptant à la syntaxe AspectJ. Nous soulignons au besoin les différences entre les possibilités actuelles avec AspectJ et un langage général de coupe sur le flot de contrôle.

À titre d'exemple pour les coupes, nous ciblons l'exécution d'une méthode `action` dans une interface `Node`. Ceci pose comme hypothèse supplémentaire l'implémentation de cette interface par tous les nœuds de la structure. Cependant cette hypothèse est facultative et nous permet surtout de simplifier l'écriture des coupes. En effet de cette façon le flot de contrôle hiérarchisé devient récursif : une coupe basique sur cette action est alors suffisante pour désigner tous les points de jonction dans le flot de contrôle, de façon indépendante des types concrets des différents nœuds.

Stratégies de parcours

La stratégie *nœuds* correspond, dans notre hypothèse simplificatrice, à une coupe basique.

```
pointcut basicStrategy(): execution(void Node+.action(..))
```

La stratégie *feuilles*, particulièrement dans une structure récursive, implique d'exclure tous les appels récursifs. Ceci utilise l'opérateur `pathabove`, lequel retourne tous les appels au-dessus du point de jonction courant, donc récursifs. Cet opérateur, dont la sémantique est définie dans [Douence et Teboul, 2004], n'est pas disponible dans AspectJ. La stratégie *feuilles* ne peut donc être exprimé dans le langage de coupes d'AspectJ.

```
pointcut leafStrategy():
    basicStrategy() && !pathabove(basicStrategy())
```

À l'image inverse des feuilles, la stratégie *sommet* exclut les exécutions récursives grâce à l'opérateur `cflowbelow`. Il faut noter que cette stratégie ne sélectionne pas forcément le sommet absolu de la structure hiérarchique, mais le sommet du flot de contrôle courant (qui peut ne concerner qu'un sous-arbre dans la structure).

```
pointcut topStrategy():
    basicStrategy() && !cflowbelow(basicStrategy())
```

La stratégie *branche* ne peut être définie qu'en ciblant le type précis (ici appelé `NodeDecorator`) subsumant la notification dans la hiérarchie d'appel de l'action.

```
pointcut branchStrategy():
    basicStrategy() && pathabove(execution(void NodeDecorator+.action(..)))
```

Capture et passage de contexte. AspectJ offre plusieurs opérateurs pour la capture dans le contexte courant. La référence à l'objet courant est obtenue par l'opérateur `this()`. Combiner cet opérateur avec la coupe du parcours est trivial et permet de passer directement le contexte avec la coupe.

```
pointcut strategyWithContext(Node context):
    strategy() && this(context);
```

Il en va différemment quand la capture du contexte, nœud parent ou enfant, est dissocié du point de notification dans le parcours. Il faut alors définir une coupe dédiée à la capture du contexte puis, si possible, combiner cette coupe de capture avec la coupe de parcours. Ceci implique que les deux points de jonction soient dans le même flot de contrôle, la capture précédent la jonction. Dans le cas contraire, il est toujours possible de séparer parcours et contexte et de stocker celui-ci dans un état intermédiaire.

Le passage de contexte d'un nœud parent vers un fils se fait dans le flot de contrôle. Dans l'hypothèse où `parentContext` définit le contexte parent, ce passage à la stratégie `strategy` s'écrit avec AspectJ :

```
pointcut strategyWithParent(Node parent):
    strategy() && cflowbelow(parentContext(parent));
```

L'utilisation de l'opérateur `cflowbelow` (au lieu de `cflow`) est nécessaire dans la mesure où l'action est récursive : `strategy` et `parentContext` ciblent la même action, distinguée uniquement par le niveau dans la pile d'exécution.

La définition de `parentContext` doit cibler un parent dans la hiérarchie. La sémantique la plus basique permet de capturer l'objet le plus récent dans la pile d'appel. Une fois injecté dans la stratégie ci-dessus, cet objet devient le parent direct puisque l'on capture le flot de contrôle conséquent avec `cflowbelow`.

```
pointcut parentContext(Node parent):
    strategy() && this(parent);
```

Il est possible par exemple de passer le nœud sommet en capturant celui-ci avec la stratégie homonyme :

```
pointcut parentContext(Node top):
    topStrategy() && this(top);
```

Contrairement au langage général présenté en [Douence et Teboul, 2004], il n'est cependant pas possible de capturer avec le langage de coupe d'AspectJ n'importe quel objet intermédiaire dans la hiérarchie de l'action.

De façon similaire, le passage de contexte d'un nœud fils vers un parent se fait dans le chemin de retour du flot de contrôle. La stratégie s'écrit alors :

```
pointcut strategyWithChild(Node child):
    strategy() && pathabove(childContext(child));

pointcut childContext(Node child):
    strategy() && this(child);
```

L'utilisation de l'opérateur `pathabove` indique que cette stratégie n'est pas exprimable directement avec le langage de coupe d'AspectJ.

6.5.2 Application aux motifs Observer, Composite et Decorator

Nous illustrons maintenant l'usage des stratégies AspectJ dans le cadre de la composition des motifs Observer, Composite et Decorator. Les motifs Composite et Decorator définissent chacun une structure récursive apte à cette application. De plus, leur composition donne une bonne modalité de cohabitation : une structure récursive peut être constituée d'instances de `Decorator` et de `Composite` entremêlées sans difficulté. Il en va différemment du motif Observer, dont les notifications sont en interactions avec la structure (section 4.3.3).

Nous examinons dans un premier temps la composition avec le seul motif Observer aspectisé (section 5.2.1). Dans un second temps, nous examinons la composition des trois motifs aspectisés. En effet l'aspectisation change les caractéristiques des motifs Composite (section 6.3.2) et Decorator (section 5.3.1).

6.5.2.1 Approche avec les solutions objets de Composite et Decorator

Les solutions objets Decorator et Composite définissent un flot de contrôle récursif pour chaque action appelée sur la structure. En effet ces actions ont pour principe de rediriger l'appel sur leurs descendants. Les stratégies ci-dessus peuvent donc être adaptées directement aux actions ciblées.

Par exemple un déplacement par l'action `move` concernera toujours l'ensemble des figures sélectionnées. Il suffit donc de déclencher la notification au sommet. Le nœud courant représente alors l'ensemble des figures concernées.

```
pointcut moveAction(): execution(void Figure+.move(..));
pointcut topMove(Figure f):
    this(f) && moveAction() && !cflowbelow(moveAction());
```

Une telle stratégie peut être définie pour plusieurs actions à la fois en adaptant la déclaration `moveAction`. Cependant il n'est pas possible de paramétrer la stratégie de façon à la réutiliser. Une libraire de stratégies contenant par exemple `topStrategy(basicStrategy)` (où `basicStrategy` est un paramètre) ne peut être par conséquent définie.

Dans le cas du motif DECORATOR, nous ne pouvons utiliser la stratégie branche en l'absence des opérateurs `path` et `pathabove`. Une sémantique similaire et correcte est possible en passant le contexte `Decorator` vers la figure concernée. Si nous considérons une chaîne d'instances de `Decorator`, alors la règle suivante se pose : un objet `Decorator` devant se substituer à l'objet décoré, l'instance au sommet se substitue à la chaîne. Notre capture du contexte vise donc l'instance `Decorator` au sommet de la chaîne d'appel avec `topDecorator` (conjonction d'une stratégie *sommet* et d'une capture locale).

```
pointcut decoratorAction():
    execution(void DecoratorFigure.setAttribute(..));
pointcut topDecorator(DecoratorFigure df):
    this(df) && decoratorAction()
    && !cflowbelow(decoratorAction());
```

Nous définissons ensuite la notification, qui doit avoir lieu dans le flot de contrôle du `Decorator`. Le contexte effectivement défini est récupéré depuis la coupe `topDecorator`.

```
pointcut decoratorNotify(Figure f):
    execution(void Figure+.setAttribute(..))
    && cflowbelow(topDecorator(f));
```

Cependant cette coupe n'est pas satisfaisante dans la mesure où elle sélectionne toutes les figures sous le sommet, y compris d'autres `Decorator` s'il existe une chaîne. Des notifications redondantes ont donc lieu. Ce défaut, dû à l'absence des opérateurs `path` et `pathabove` pour sélectionner les feuilles, peut être corrigé au cas par cas en restreignant le type de la figure à un type feuille (comme `LineFigure` ci-dessous), excluant les types récursifs. La coupe perd alors sa généricité.

```
pointcut decoratorNotify(Figure f):
    execution(void LineFigure+.setAttribute(..))
    && cflowbelow(topDecorator(f));
```

Les conséquences de l'utilisation de ces coupes en terme d'impact et d'expressivité sont remarquables. Les interactions sont traitées de façon modulaire côté `Observer`. De plus il est possible d'obtenir un grain fin en ciblant certaines actions avec des stratégies spécifiques.

6.5.2.2 Approche avec les solutions aspects de Composite et Decorator

L'aspectisation des motifs de conception transforme leurs implémentations et donc une partie de leurs caractéristiques. Or ces caractéristiques définissent les hypothèses sur lesquelles s'appuient nos stratégies d'observation. Nous examinons donc les conséquences de l'aspectisation de Composite et Decorator sur l'expression des stratégies.

L'intérêt de l'aspectisation du motif Composite est sa propriété réutilisable : nous utilisons donc le motif présenté en section 6.3. Cette solution ne change pas la caractéristique récursive du motif. En particulier il est toujours possible d'effectuer des actions récursives sur la structure (voir méthode `recurseAll` du code source 6.1). Cependant la récursion implique le motif Visitor pour appeler l'action. Il faut donc paramétrer les coupes en conséquence.

La coupe `action` cible la méthode de récursion générique `recurseAll` de l'aspect `CompositeProtocol`. Le passage de contexte ne change pas mais la paramétrisation précise de l'action visée se fait dans la stratégie `topMove` : il s'agit de filtrer les instances d'une occurrence `MoveVisitor` dans le second argument de `recurseAll`.

```
pointcut action():
    execution(void CompositeProtocol.recurseAll(..));
pointcut topMove(Figure f):
    args(f,MoveVisitor) && action() && !cflowbelow(action());
```

Au contraire de Composite, le motif Decorator aspectisé ne fonctionne pas par récursion. Il ne modifie donc pas le flot de contrôle récursif et est invisible aux stratégies fondées sur cette caractéristique, tel *sommet*, *feuilles* et *branche*. Ceci est aussi gênant pour la récupération et le passage de contexte : l'objet décoré est transmis dans la notification hors du contexte de Decorator.

Cependant l'utilisation du motif Decorator aspectisé dans ce cas n'est pas forcément incorrecte. En effet la définition du contexte de notification par la seule référence à l'objet observé implique que l'information adéquate est récupérée ultérieurement par interrogation de celui-ci. Or ces appels ultérieurs sont interceptés par l'aspect `Decorator` qui effectue alors son action.

Cette aspectisation de Decorator a donc comme caractéristique de retarder l'interaction au dernier moment, contrairement aux stratégies qui procèdent par anticipation.

6.5.3 Bilan

Cette section montre que les interactions complexes ne se résolvent pas par rapport à un point mais un contexte. Dans ce cas la résolution dépend de la définition du contexte et du pouvoir d'expression des coupes vis-à-vis de ce contexte.

Dans ce cadre nous avons montré comment le langage de coupe d'AspectJ sur le flot de contrôle permet de résoudre des interactions dans le cas d'une structure d'objet hiérarchisée, potentiellement récursive. Ce cas s'applique en particulier à l'interaction des motifs Observer, Composite et Decorator.

Cependant les caractéristiques actuelles du langage AspectJ amènent deux remarques. La première est que celui-ci ne propose pas un langage complet pour résoudre ces interactions. La seconde est que l'hétérogénéité du traitement entre instances d'objet et instances d'aspect met en difficulté le raisonnement hiérarchique : il est parfois nécessaire d'adapter ce raisonnement au point de vue de l'aspect.

Retour sur la transformation du motif Observer. Dans ce problème l'interaction des solutions objets aboutissait à la transformation du motif Observer en Chain of Responsibility (section 4.3.3). Nous constatons que la double récursion imbriquée dans la structure (section 4.3.3.3) de l'action (descendante) et de la notification (ascendante) est remplacée par la seule récursion descendante de l'action, augmentée par les aspects aux points d'entrée (`call`) et de sortie (`return`) de chaque action.

De plus le choix d'utiliser le motif Chain of Responsibility est une solution à gros grain (section 4.3.3.4) : la stratégie *branche* était appliquée à toutes les notifications sans distinction. Le langage de coupe permet de cibler chaque action avec un grain différent.

6.6 Bilan

Nous disons qu'il y a composition de motifs dès que les rôles de deux occurrences différentes se superposent sur une seule classe. L'implémentation modulaire des motifs avec les aspects ouvre la voie à l'étude de la composition des motifs dans le code. Notre étude couvre deux niveaux d'abstraction : un niveau abstrait de composition avec le concept de modalité pour décrire les compromis de la modularité. Un niveau concret avec la présentation des mécanismes d'AspectJ pour la composition structurelle et comportementale. Dans cette conclusion, nous tirons les leçons à exploiter par le programmeur.

6.6.1 Modularité et modalité de composition

En fonction du langage de programmation et des spécifications du code, le programmeur peut aboutir à un compromis entre modularisation et invasion de code pour implémenter une composition. Nous proposons le concept de modalité pour désigner et discuter le degré de modularité attendu (soit aussi, le degré d'imbrication autorisé) dans la composition de deux motifs.

D'une manière générale, la composition de motifs se réduit à la composition des éléments d'implémentation de ces motifs. Il en résulte un compromis entre la modularité visée et des choix d'implémentation simplificateurs et pragmatiques. En particulier un bon compromis consiste à implémenter une composition asymétrique, c'est-à-dire où un seul des motifs fait directement référence à l'autre. C'est le cas dans nos exemples du motif Composite avec Visitor et du motif Observer avec Composite et Decorator. Ces compromis n'empêchent pas de définir des compositions réutilisables.

6.6.2 Mécanismes de composition et de résolution d'interactions

Nous examinons en détail les mécanismes de composition structurelle et comportementale d'AspectJ, en particulier pour la résolution d'interactions et de conflits.

La composition structurelle repose sur les déclarations intertypes d'AspectJ. Nous comparons et soulignons la proximité entre les déclarations intertypes et les traits. Cependant le modèle des traits offre une plus grande finesse dans la résolution des conflits grâce à ses opérateurs. En vue d'une intégration de ces opérateurs à AspectJ, nous décrivons deux solutions possibles pour le typage statique des traits.

La composition comportementale peut s'opérer avec le langage de coupe des aspects. Nous montrons la résolution d'interactions dans une composition de motifs à l'aide des coupes sur le flot de contrôle. Cet exemple illustre qu'il est parfois nécessaire de s'affranchir du contexte

des classes pour résoudre des interactions de façon élégante. D'une manière générale, l'utilisation d'AspectJ en complément de Java offre une meilleure expressivité pour un faible coût d'implémentation.

Souplesse de la composition comportementale. Du point de vue des langages de programmation, il existe une différence notable entre la souplesse de la composition comportementale permise par AspectJ et la rusticité des déclarations intertypes. La gestion des conflits est limitée avec les déclarations intertypes par défaut d'opérateurs de composition.

À l'inverse, différents exemples illustrent comment les coupes et les aspects permettent de définir une liaison tardive par rapport à différentes caractéristiques des points de jonction.

- La solution Visitor est générique grâce au filtrage sur les signatures, ce qui facilite aussi sa composition avec le motif Composite réutilisable.
- Les interactions comportementales du motif Observer sont résolues par des coupes sur le flot de contrôle au lieu d'une transformation statique en motif Chain of Responsibility.
- Le motif Decorator aspectisé interagit de façon transparente avec Observer (section 6.5.2.2).

Chapitre 7

Méthode de programmation avec AspectJ

Sommaire

Nous capitalisons les résultats obtenus dans l'aspectisation et la composition des motifs de conception dans une méthode de programmation avec AspectJ. Cette méthode s'applique en particulier à la restructuration d'un programme objet par les aspects. Elle guide la démarche du programmeur à travers la sélection et la mise en œuvre des préoccupations à aspectiser. Les critères associés à la méthode dépendent du domaine, comportemental ou structurel, visé par la démarche du programmeur. Nous illustrons et commentons les résultats de cette méthode appliquée à JHotDraw.

Cette méthode a aussi été conçue dans le but d'évaluer la pertinence et la faisabilité de notre approche, ainsi que pour exposer nos contributions à la programmation par aspects.

7.1 Introduction

Notre méthode de programmation est basée sur l'aspectisation des motifs ainsi que l'utilisation des compositions potentielles. Elle peut cependant s'appliquer à toute préoccupation d'un programme visant une meilleure modularité par les aspects.

Elle se veut un guide pratique pour appliquer la programmation par aspects, en particulier dans le cas d'une restructuration d'un programme objet. Nous décomposons donc notre méthode en deux étapes : la sélection et la mise en œuvre. Pour chaque étape un ensemble de critères adéquats est proposé. Notre démarche n'est cependant pas complète ni linéaire : le programmeur peut, de ce fait, n'en utiliser qu'une partie ou encore réitérer les étapes.

La sélection consiste à utiliser des critères d'application pour juger si l'aspectisation est pertinente. Si cette étape est positive, la mise en œuvre (conception suivie de l'implémentation) permet de décomposer et décrire l'aspect, sa structure et en particulier son interface transversale.

L'état des techniques montre que les domaines structurel et comportemental requièrent deux approches différentes. Cette différence se répercute aussi sur les critères de notre méthode. De même que les éléments des deux domaines se complètent, l'usage de ces critères est complémentaire. Notre démarche ne prétend pas cependant pas être exhaustive quant aux bonnes pratiques des aspects.

7.2 Description de la méthode

Les sept étapes données ci-dessous forment une vue d'ensemble de la démarche. Nous distinguons entre des critères généraux, valables quel que soit le contexte, et des critères conjoncturels dont l'évaluation n'est pas toujours nécessaire. La hiérarchie ci-dessous donne un aperçu général de la démarche et de ces critères : l'objectif est d'arriver à une décision en considérant d'abord les critères plus généraux avant de se focaliser sur les particularités du contexte et les spécificités du langage d'aspect. Il n'y a cependant pas de règle absolue et la décision relève du faisceau d'indices dégagé par les critères.

Dans notre approche les critères généraux sont les premiers à évaluer pour décider de l'aspectisation.

1. Identification du module préoccupation-motif.
2. Discrimination des domaines entre classes et aspects.
3. Détection des symptômes d'aspect conceptuel : dispersion et mélange des préoccupations, couverture des interfaces.

Les critères conjoncturels forment un ensemble plus disparate, à appliquer au cas par cas.

4. Adéquation des propriétés recherchées du motif au potentiel offert par le langage d'aspect.
5. Besoin d'adaptation (polymorphisme).
6. Besoin de variabilité de la préoccupation.

7.2.1 Identification du module préoccupation-motif

Nous défendons la règle suivante : l'aspectisation englobe la préoccupation et son motif support. L'objectif est, suivant le critère de masquage de l'information [Parnas, 1972], d'isoler dans l'aspect les choix de conception du motif avec la préoccupation liée.

Cette démarche a aussi pour conséquence de rejeter, a priori, les collaborations de classe pour implémenter la préoccupation. Par la suite, les différents critères expliquent comment définir/rétablir ces collaborations avec les classes : ceci concerne en particulier la séparation des domaines et le polymorphisme.

Par conséquent nous suivons une modularisation classique, en particulier en distinguant interfaces requises et fournies. Le programmeur doit étudier les deux interfaces de son aspect :

- le contexte requis par son aspect ;
- l'usage de son aspect avec et par d'autres modules.

Il est possible de faire appel aux méthodes de l'interface de l'aspect. L'aspect est alors utilisé comme une classe normale. De même, si un aspect se caractérise par l'usage de coupes pour se brancher aux autres modules, celui-ci peut aussi utiliser l'interface classique de ces modules.

Notre approche n'implique donc pas une dichotomie entre une base classique et une couche d'aspects tissés a posteriori, mais une intégration transparente en tant que module. Les aspects sont alors utilisés pour inverser les dépendances et établir une hiérarchie acyclique.

7.2.2 Discrimination des domaines entre classes et aspects

Une deuxième règle générale que nous nous sommes fixée est la discrimination des domaines entre modules. Ceci correspond en particulier au fait que les mécanismes d'AspectJ (coupes-actions et déclarations intertypes) ont des usages et des cibles différents. Il s'agit donc de choisir la technique disponible la plus adaptée au contexte.

Cette règle se traduit en pratique par une plus grande séparation des rôles entre modules. En Java ces différents rôles sont joués par les seules classes. Nous discriminons les rôles suivant :

- déclaration, définition et instantiation d'une structure de données et de ses comportements : classes ;
- définition d'unités de réutilisation et de collaborations configurables par les classes : traits (aspects structuraux).
- définition de comportements et collaborations externes aux classes : aspects de comportement.

La notion de collaboration externe aux classes, par opposition aux collaborations des traits, fait appel à deux notions différentes de polymorphisme que nous détaillons section 7.2.5. Une conséquence importante de cette distinction, et en pratique la tâche la plus difficile, est qu'il faut parfois « oser » casser le polymorphisme d'héritage pour aspectiser une préoccupation.

Cette règle générale n'est pas pour autant absolue. Un aspect peut mélanger les deux domaines. Le module aspect peut utiliser à la fois comportement par coupe-action et collaborations de classe par déclarations intertypes.

7.2.3 Détection des symptômes d'aspect comportemental et structurel

Les deux règles générales précédentes décrivent l'intégration des aspects dans une démarche de modularisation. La détection dans l'implémentation des symptômes liées aux préoccupations transversales est un autre facteur de décision, plus spécifique aux aspects. Nous expliquons ci-dessous différents moyens et méthodes utilisés pour détecter et évaluer ces symptômes.

Ces méthodes passant par l'identification de portions de code à travers l'implémentation, elles sont aussi utiles pour extraire ce code dans l'aspect lors de la restructuration. La détection des symptômes est donc une étape essentielle dans la sélection et la mise en œuvre.

Nous avons séparé les symptômes suivant les deux domaines, partant du principe que ceux-ci ont un objectif différent. Ainsi dans le domaine comportemental nous nous intéressons surtout à la dispersion et au mélange des préoccupations. Dans le domaine structurel, nous nous intéressons à la réutilisation de code et plus spécifiquement aux mauvais usages de l'héritage (section 4.3.4.2 page 71).

7.2.3.1 Aspect comportemental

Il s'agit d'évaluer deux symptômes signant la présence de préoccupations transversales : la *dispersion* et le *mélange* des implémentations de ces préoccupations.

La dispersion affecte deux caractéristiques d'une implémentation : les éléments proprement dits de l'implémentation et les références à ces éléments. Tracer la dispersion des éléments d'une préoccupation implique d'abord d'identifier ces éléments à travers le code. Nous ne connaissons pas de méthode permettant leur détection : l'identification se fait manuellement par parcours du code et jugement de l'affinité avec la préoccupation. Par contre, une fois ces éléments identifiés, il existe plusieurs techniques pour tracer les références dispersées de ces éléments.

La dispersion des références doit être évaluée suivant deux axes : le nombre de références à un élément de la préoccupation et le nombre de modules où ces références sont faites. Si le score est grand sur ces deux axes, le cas de dispersion est clair. Sinon le cas doit être examiné en détail pour déterminer si la dispersion justifie l'aspectisation (en particulier pour une meilleure modularité).

La dispersion peut être estimée de façon partielle et empirique avec les outils des environnements de développement intégrés (EDI). Il s'agit de chercher et tracer les références aux éléments de la préoccupation étudiée : ces références sont les appels de méthode, les accès aux variables, les déclarations de types. Les appels de méthode présentent un cas particulier car la dispersion peut être initialement masquée par des indirections : une référence unique au module transversal peut être utilisée dans une seule méthode – mais les appels à cette méthode d'indirection sont dispersés à travers le code.

L'EDI *Eclipse* fournit des outils de recherche des références aux éléments et de traçage des appels de méthode : il faut lancer la recherche manuellement pour chaque élément. Le traçage des appels de méthode est particulièrement intéressant car il peut se faire de façon récursive (chercher l'appelant de l'appelant de...). Ceci permet de détecter plus facilement les appels dispersés malgré les indirections.

Une autre méthode utilise directement AspectJ et les outils de visualisation des coupes. Elle consiste à définir un simple aspect de trace dont les coupes capturent les références aux éléments ciblés. L'avantage de cette méthode est d'obtenir et d'enregistrer des visualisations complètes de ces traces.

Le mélange est plus délicat à juger puisqu'il implique d'analyser finement une portion de code : chaque expression doit être associée à la préoccupation intéressée. Le mélange demande donc une analyse et une compréhension complète du module courant et de ses dépendances. Cette démarche est identique à l'identification des éléments d'une préoccupation pour évaluer leur dispersion.

L'estimation du mélange compte le nombre de préoccupations et rapporte ce nombre à la taille du code : cependant cette mesure n'est pas absolue et nous préférons nous rapporter au

critère de masquage d'information pour juger si le mélange est approprié.

Nous n'avons donc pas utiliser de cadre général pour évaluer le mélange. Cependant l'utilisation d'outils de détection et de traçabilité est à examiner.

7.2.3.2 Aspect structurel

Les problèmes de réutilisation explorés dans notre thèse (section 4.3.4.2 page 71) sont d'une nature particulière. Il s'agit de cas où la réutilisation est forcée, ceci car l'héritage simple est souvent le seul mécanisme disponible à cet usage. Nous nous intéressons ici à ce second problème dans le cadre de Java et de ses interfaces.

Notre méthode d'analyse repose sur la comparaison (ou couverture) des interfaces de deux modules T et S (interface Java ou classe) dans une relation ordonnée $S > T$ (S est considéré comme un supertype de T). Cette relation de sous-typage n'existe pas nécessairement dans le code : il est tout à faire possible, avec quelques précautions d'interprétation, de comparer deux modules sans relation (bien que notre vocabulaire suppose cette relation).

Nous définissons grâce à cette comparaison ordonnée trois catégories de méthodes. Ces catégories sont définies du point de vue de T par des opérations entre les deux ensembles de méthodes des interfaces I_T et I_S.

- Méthodes héritées ($M_i = I_S - I_T$) : méthodes de S directement héritées par T.
- Méthodes redéfinies ($M_o = I_S \cap I_T$) : méthodes de S redéfinies par T.
- Méthodes nouvelles ($M_n = I_T - I_S$) : méthodes absentes de S et définies par T.

La catégorie des méthodes redéfinies peut être raffiner après un examen de leur implémentation en deux sous-catégories.

- Méthodes spécialisées M_s : méthodes de T faisant appel à la méthode correspondante cachée dans S via `super`.
- Méthodes remplacées M_r : méthodes de T ne faisant pas appel à `super`.

L'interprétation de cette comparaison se base sur les quatre catégories des méthodes héritées, nouvelles, spécialisées et remplacées. Nous comparons la taille de ces différents ensembles pour évaluer la part de réutilisation transparente (proportion de méthodes héritées), la nouveauté et la spécificité du module T (proportion de méthodes nouvelles et spécialisées), ou encore l'inadéquation de l'héritage (proportion de méthodes remplacées).

Surtout, ces mesures sont assez simples et souples pour comparer des interfaces sans relation directe dans un graphe d'héritage et d'implémentation d'interfaces Java. Il est par exemple possible de comparer T avec l'arbre d'héritage complet de S. En analysant ces mesures, nous détectons quelles interfaces sont implémentées par une classe, même s'il n'y a pas de relation de sous-typage directe ou transitive entre les deux.

7.2.4 Adéquation des propriétés du motif aspectisé au problème

Comme montré au chapitre 5, tous les motifs aspectisés ne permettent pas la réutilisation et sont plus ou moins simples à implémenter. Plus encore, certains motifs aspectisés n'ont pas les mêmes propriétés que leurs homologues objet. Il est donc prudent avant d'aspectiser un motif de vérifier si les propriétés requises par le contexte sont possibles avec AspectJ. Le motif Decorator, qui est plus simple que son pendant objet mais ne permet pas la composition récursive, est emblématique de ces cas tangents.

Il n'existe pas (encore) de catalogue des motifs aspectisés et de leurs propriétés. Nous nous contentons ici de citer les cas problématiques déjà connus, en attendant que l'expérience apporte de nouvelles contributions.

La gestion des instances sous AspectJ étant contrainte, il est parfois délicat de traiter un rôle dynamique directement dans un aspect (malgré les idiomes du chapitre 5). Ceci est le cas pour le motif DECORATOR, certains usages du motif PROXY, la version objet du motif ADAPTER.

De même, le mécanisme de coupe–action permet de capturer un point de jonction mais pas d'en modifier certains attributs, comme le receveur. Il est donc impossible de rediriger un appel de méthode (sans passer par la réflexion). En pratique plusieurs motifs sont affectés par ce syndrôme des signatures spécifiques, où le comportement du motif est basé sur la redirection de méthodes spécifiques à chaque occurrence (sections 5.4.2.1, 5.4.2.3 et 5.4.3). Ce syndrôme gêne l'abstraction générique d'un comportement par une coupe-action. Outre ceux cités dessus, il peut s'agir des motifs structurels comme BRIDGE ou FACADE.

7.2.5 Adaptation et polymorphisme

Une des conséquences les moins évidentes de notre travail est la place du polymorphisme avec AspectJ. En effet la localisation du code dans les aspects casse les collaborations de classe, donc la capacité d'adaptation de celles-ci grâce au polymorphisme d'héritage. Localiser dans un aspect une adaptation spécifique à une classe peut conduire à une violation du masquage d'information et de la modularité.

Nous résolvons ce problème en discriminant les usages du polymorphisme.

- Le polymorphisme ad hoc indique que l'aspect peut décider seul du comportement à appliquer en fonction du contexte, sans violer la modularité d'autres classes.
- Le polymorphisme d'héritage indique au contraire que l'adaptation dépend de la classe visée : l'aspect fait donc toujours appel à une méthode de la classe pour adapter le comportement.

La programmation par AspectJ est donc moins intéressante si le motif repose fortement sur le polymorphisme d'héritage. Cependant il est aussi possible d'obtenir un polymorphisme d'héritage grâce aux déclarations intertypes. Ce procédé vient brouiller les limites entre aspect structurel et aspect comportemental.

Enfin l'utilisation de coupes relationnelles montre que les aspects changent aussi la définition du contexte de polymorphisme. La section 6.5 montre comment en AspectJ l'adaptation de l'aspect se fait par rapport aux relations entre instances à l'aide des coupes sur le flot de contrôle, plutôt que sur la seule signature des méthodes.

7.2.6 Variabilité

Enfin une dernière conjoncture à prendre en compte est la variation des implémentations due à différentes configurations (voir les conséquences du motif SINGLETON en section 4.3.2). Dans ce cas la motivation est simplement d'aspectiser la préoccupation pour offrir le choix de l'aspect adéquat au tissage.

7.2.7 Critères de conception et d'implémentation

La démarche de conception et d'implémentation s'appuie sur les critères précédents pour identifier les parties à aspectiser et les techniques à utiliser. En particulier l'identification des

domaines, la détection des symptômes et le choix du polymorphisme fixent les spécifications.

Cependant d'autres choix peuvent survenir durant la phase de conception. Nous nous appuyons en particulier sur les idiomes et remarques exposés au chapitre 5 pour résoudre ces choix.

Un choix important est l'implémentation de la relation entre les rôles des motifs. Suivant la cardinalité de la relation, il peut être intéressant de déclarer une instance d'aspect liée (section 5.2.1.4) ou bien d'utiliser les idiomes de structure (section 5.2.2.2). Il est bien sûr possible de réutiliser les motifs génériques, définies avec des relations n, n (section 5.4.2).

Enfin la déclaration des coupes peut amener de nouveaux problèmes : nous avons parfois dû adapter celles-ci pour accomoder le contexte. En particulier un point auquel les coupes et aspects d'AspectJ sont fragiles est la prise en compte de l'état des objets ciblés : il arrive qu'une coupe capture un point de jonction où l'objet capturé est dans un état inconsistant (par exemple dans le constructeur). Ceci entraîne souvent une violation du masquage d'information car il faut accéder au contexte particulier des objets pour traiter ces cas.

7.3 Restructuration de JHotDraw

Nous présentons les résultats de notre démarche sur le cadriciel JHotDraw. L'objectif n'est donc pas d'effectuer une aspectisation complète du programme mais de sélectionner les candidats les plus intéressants. Cette démarche se veut donc une démonstration pragmatique de l'aspectisation.

Comme au chapitre 4, nous avons limité le contexte de l'étude à une version basique de JHotDraw, correspondant au noyau et aux extensions `CompositeFigure` et `DecoratorFigure`.

7.3.1 Motifs Observer

Nous avons isolé six préoccupations supportées par des occurrences du motif Observer dans le cadriciel JHotDraw original. Ces occurrences sont en particulier identifiées par l'usage du suffixe `Listener` pour nommer l'interface Java de l'observateur. Cette interface permet d'abstraire les références à l'observateur dans le sujet. Le but général de ces diverses occurrences est le maintien de la cohérence entre le modèle du dessin, les vues et les commandes. Nous discutons ici de la décision d'aspectiser ces préoccupations et des choix de conception remarquables.

7.3.1.1 Aspectisation de `FigureChangeListener` et `DrawingChangeListener`

Cette préoccupation étant explorée dans plusieurs de nos travaux (sections 4.3 et 6.5.2), nous donnons simplement un aperçu du but et des efforts demandés par ce travail.

Intention. Ces deux occurrences forment le support de la préoccupation d'invalidation vue en section 4.3.1. Cette préoccupation participe à la préoccupation plus générale de rafraîchissement de la vue après les modifications du dessin. Le rafraîchissement est aussi supportée par un mécanisme de requête de mise à jour.

Le rafraîchissement se déroule donc en deux étapes :

1. invalidation des `Figure`s et accumulation des zones invalidées ;
2. requête de mise à jour de la vue, avec mise à zéro de la zone accumulée.

La décision est basée sur les critères généraux.

- Le rafraîchissement est une préoccupation aux caractéristiques internes assez complexes pour être cachées.
- Les notifications d'invalidation et les requêtes de mise à jour sont dipersées en de nombreux points sur plusieurs classes.
- `FigureChangeListener` mélange la préoccupation d'invalidation avec un mécanisme de notification générique entre figures.

Outre la dispersion de cette préoccupation sur de nombreuses classes, la section 4.3.3 montre comment les interactions du motif OBSERVER avec les motifs COMPOSITE et DECORATOR impliquent une transformation de motifs et des modifications invasives de la préoccupation.

Conception. La modularisation de cette préoccupation est raffinée par trois aspects : `DrawingDamage`, `GetDrawingDamage` et `RepairDrawingDamage`.

`DrawingDamage` est un aspect structurel définissant pour chaque dessin `Drawing` une zone de cumul temporaire d'invalidation, ainsi que les méthodes d'accès associées. Il remplit à la place de `Drawing` le rôle d'observateur dans la préoccupation d'invalidation ci-dessous.

`GetDrawingDamage` est l'aspectisation des deux occurrences `FigureChangeListener` et `DrawingChangeListener` en charge de l'invalidation. Plus exactement, cette aspectisation a trois principales caractéristiques :

- chaque `Figure` est associée à un unique dessin, d'où une cardinalité unaire dans la relation Sujet-Observateur.
- notification de l'aspect `DrawingDamage` associé au dessin observateur.
- application des stratégies d'observation (section 6.5.2) pour capturer le contexte adéquat.

Par ailleurs, la restructuration a entraîné les choix suivant :

- séparation entre des interfaces d'invalidation et de notification dans `FigureChangeListener` – seule la première préoccupation est traitée, la seconde étant accessoire.
- capture des points d'invalidation – ceci inclut de nombreux appels indirects.
- élimination des diverses occurrences TEMPLATE METHOD dont le but était d'abstraire ces notifications.

`RepairDrawingDamage` extrait la préoccupation de mise à jour proprement dite. Celle-ci apparaît dans le cadriciel original comme une occurrence cachée du motif OBSERVER. Sa fonction est de récupérer (et remettre à zéro) la zone invalidée pour lancer la procédure de mise à jour Swing.

Nous sommes ainsi passé des deux interfaces, des implémentations dispersées et d'un motif caché à trois aspects, chacun correspondant à une partie de la préoccupation principale. L'implémentation des classes auparavant directement concernées par cette préoccupation est aussi soulagée : par extraction du code des motifs mais aussi par élimination de `Template Methods` servant à cacher l'implémentation.

7.3.1.2 Aspectisation de `FigureSelectionListener`

Un principe imposé par le cadriciel JHotDraw est la distinction entre outils et commandes pour la manipulation des figures. Les outils agissent directement dans les vues du dessin pour créer, manipuler ou sélectionner des figures avec le curseur. Les commandes, accessibles par les menus, manipulent (groupent, alignent ...) la *sélection* de figures (faite préalablement avec l'outil de sélection).

Intention. Une conséquence de ce principe est que les commandes peuvent être activées ou désactivées en fonction de la sélection. Si par exemple la sélection est vide, la plupart des commandes sont inopérantes. Le motif OBSERVER associé à `FigureSelectionListener` permet la notification des changements de sélection afin de vérifier l'activation des commandes.

Décision. L'activation des commandes est une préoccupation simple et indépendante. Son implémentation (légère) impacte `DrawingView`, `DrawingEditor` et `CommandMenu` (menu de commandes), ainsi que les `Command`s. Les points de notification (changement de la sélection dans la vue) sont limités et localisés dans `DrawingView`.

Néanmoins, le motif supportant cette préoccupation reste dispersé entre `DrawingView` (sujet), `DrawingEditor` (observateur) et `CommandMenu` en tant que cible finale de la notification. Du fait de cette dispersion et de l'indépendance de la préoccupation, nous avons décidé d'extraire le motif des entités `DrawingView` et `DrawingEditor` dans un aspect.

La conception de JHotDraw fait que l'ensemble des menus est unique et associé au singleton `DrawingEditor` : un aspect singleton peut donc remplacer le rôle observateur de `DrawingEditor` pour notifier directement les `CommandMenu`s. Les caractéristiques du motif sont les suivantes :

- chaque `CommandMenu` créé est enregistré en tant qu'observateur par l'aspect ;
- chaque changement de sélection dans `DrawingView` est capturé par une coupe ;
- un changement de sélection déclenche la notification :
- la notification fait appel à une méthode des `CommandMenu`s (polymorphisme d'héritage).

7.3.1.3 Aspectisation de `ToolListener`

Intention. Les outils de manipulation des `Figure`s peuvent être dans différents états d'activation, fonctions du contexte (*enabled, usable, activated*). À la façon des menus pour les commandes, les boutons d'activation des outils sont intéressés par ces changements d'état pour mettre à jour leur propre état (activé ou non).

Décision. L'implémentation objet amène à la définition d'une classe `EventDispatcher`, interne à `Tool`, pour définir le rôle sujet. Les notifications sont clairement localisées aux accesseurs de changement d'état dans `Tool`. Notre décision d'aspectiser est uniquement motivé par la simplification de `Tool` par l'extraction de la classe `EventDispatcher`.

Conception. Les classes associées aux deux bouts de cette préoccupation, `Tool` et `ToolButton`, définissent chacune leur propre ensemble d'états abstraits. Chaque instance de `ToolButton` étant liée à une instance de `Tool` (et réciproquement), le motif OBSERVER sert à assurer la cohérence entre ces deux états.

Cependant pour éviter la confusion entre les deux états abstraits, nous ignorons dans l'aspectisation celui de `ToolButton`. L'aspect détecte et notifie à `ToolButton` les états abstraits de `Tool` via l'interface `ToolListener` : la gestion de l'état d'un bouton n'est donc pas prise en charge par l'aspect mais laissée à la classe `ToolButton` par polymorphisme d'héritage.

Contrairement aux cas précédents qui visaient la simplification de la préoccupation, cette aspectisation conserve avec l'interface `ToolListener` les spécifications de l'implémentation objet. Ce choix se justifie donc par la séparation des abstractions d'états. Il illustre un aspect non omnipotent, en tant que module intégré à d'autres modules.

7.3.1.4 Non aspectisation de `PaletteListener`

Intention. La classe `PaletteButton` sert d'intermédiaire entre le cadriciel *Swing* et la classe `ToolButton`. Son rôle est de translater les événements souris transmis par *Swing* à `ToolButton` (via `PaletteButton`) en événements de l'interface `PaletteListener`, implémentée par `DrawingEditor`. `DrawingEditor` est ainsi informé des boutons pressés et, indirectement, de l'outil sélectionné.

Décision. Ce rôle de translation et notification est le seul joué par `PaletteButton`. Il n'y a pas de dispersion dans le code source de JHotDraw. Cette occurrence n'est donc pas un candidat intéressant.

7.3.1.5 Préoccupations temporairement exclues

Notre application à JHotDraw vise à isoler la base du cadriciel. Durant ce processus nous avons donc exclues des préoccupations non nécessaires imbriquées dans le cadriciel original. Ceci concerne les occurrences du motif Observer suivantes :

- la gestion des dépendances entre `Figure`s connectées, auparavant mélangée avec l'invalidation ;
- la notification des changements de vue par `ViewChangeListener` – cette préoccupation n'est nécessaire qu'avec l'extension MDI (*Multiple Document Interface*, voir aussi section 4.3.2) de JHotDraw.

Ces préoccupations font donc partie des options variables du cadriciel. Elles peuvent être réintroduites si besoin grâce aux aspects. Leur exclusion permet de simplifier la base du cadriciel.

7.3.1.6 Bilan

Le bilan de l'aspectisation des motifs Observer dans JHotDraw est une illustration de plusieurs points de notre démarche. Sur les six occurrences détectées, cinq ont été aspectisées. Cependant l'aspectisation d'un motif ne consiste pas simplement à extraire son implémentation : nous avons reconsidéré chaque préoccupation afin de procéder à une nouvelle modularisation. Le résultat est synthétisé dans le tableau 7.1.

Tableau 7.1 – Bilan de l'aspectisation des six occurrences du motif Observer dans le cadriciel JHotDraw original.

occurrence	Résultat
`FigureChangeListener`	séparation entre invalidation et connection, aspectisée
`DrawingChangeListener`	aspectisée avec l'invalidation
`FigureSelectionListener`	aspectisée avec la validation des menus
`ToolListener`	aspectisée avec la validation des boutons
`PaletteListener`	non aspectisée
`ViewChangeListener`	aspectisée avec l'option MDI

Une analyse plus fine des caractéristiques de chaque occurrence montre la diversité des relations entre sujets et observateurs : étant donné la modularité des nouvelles solutions, nous

avons spécialisé et simplifié l'implémentation de chaque aspect en fonction de la cardinalité de la relation (suivant le principe d'économie de notre noyau – section 4.2.1). Une autre option consiste à réutiliser les solutions génériques (section 5.2.1).

Le problème de conception le plus délicat auquel nous nous sommes confronté est la définition des spécifications d'un aspect, point équivalent à définir précisément la préoccupation implémentée par l'aspect. Ce problème s'est posé en particulier pour `ToolListener` où notre première conception mélangeait les deux abstractions d'état. Dans ce cas le masquage d'information et le rôle du polymorphisme d'héritage sont deux concepts qui nous ont aidé à séparer les préoccupations.

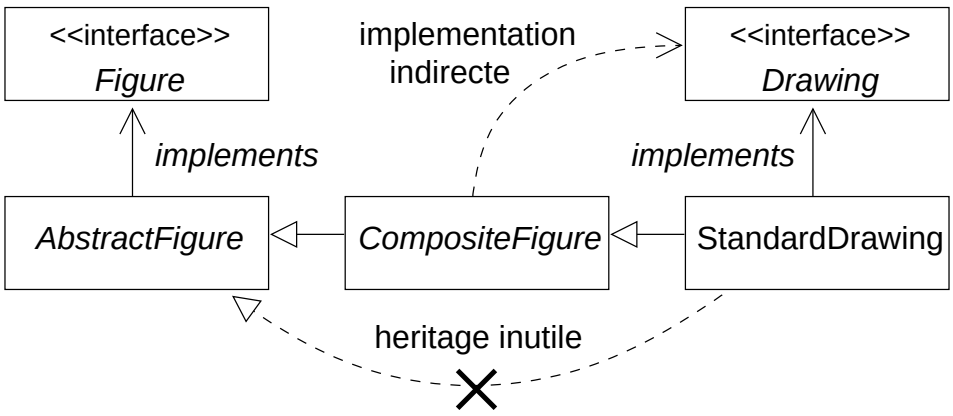
7.3.2 Motifs Composite et Visitor

Le but de l'aspectisation du motif COMPOSITE est d'en faire un module indépendant et réutilisable. Dans le cas de JHotDraw ceci revient à extraire la classe `CompositeFigure` de la hiérarchie implémentant l'interface `Figure`. Cette extraction a aussi pour objectif de simplifier l'arbre d'héritage de `StandardDrawing` en enlevant sa dépendance à la hiérarchie de `Figure` (voir section 4.3.4.2).

Ce double objectif est motivé par les intuitions suivantes (illustrées par la figure 7.1, reprise de la figure 4.8 page 71) :

1. la classe `CompositeFigure` sert d'implémentation indirecte à l'interface `Drawing` ;
2. les méthodes et attributs héritées de `Figure` par `StandardDrawing` sont inutiles.

Illustration 7.1 – Hiérarchie de `StandardDrawing` et réutilisation du motif COMPOSITE dans JHotDraw.



7.3.2.1 Intention

L'implémentation du motif COMPOSITE dans la classe abstraite `CompositeFigure` vise la manipulation des groupes de `Figure`. Par nature, elle imite l'interface de `Figure` pour permettre un usage uniforme. Les classes `GroupFigure` et `StandardDrawing` héritent toutes les deux de `CompositeFigure` mais pour des usages différents.

`GroupFigure` est une extension standard de `CompositeFigure` et, en tant qu'occurrence du motif COMPOSITE, permet la composition récursive de plusieurs `GroupFigure`.

Par contre il n'y a pas de spécifications dans JHotDraw sur la composition récursive de plusieurs `Drawing`. `StandardDrawing`, qui implémente `Drawing`, n'a donc pas besoin de la propriété récursive du motif COMPOSITE. Elle hérite néanmoins de `CompositeFigure` pour profiter des fonctions de manipulation des `Figure`.

7.3.2.2 Décision

Nous utilisons l'analyse de couverture des interfaces (section 7.2.3.2) entre les différents modules de la figure 7.1 pour démontrer l'intuition et clarifier les relations entre `StandardDrawing` et `Figure`.

Présentation des tableaux. Chaque tableau expose de façon synthétique la couverture de S par T avec l'hypothèse $S > T$ (S est un super-type de T) : la taille et la fraction de chaque catégorie sont données suivant les détails ci-dessous. Ces relations découlent des définitions des catégories décrites en section 7.2.3.2.

- Chaque type est suivi (entre parenthèses) par le nombre de méthodes ($|I_S|$, $|I_T|$) *explicitement* déclarées dans son interface.
- Le nombre *total* de méthodes disponibles dans T ($|M_t|$) correspond au cardinal de l'union des deux ensembles :
$$|M_t| = |I_S \cup I_T| = |M_i| + |M_n| + |M_o|$$
- Les pourcentages sont données par rapport à ce total.
- Les relations suivantes sont respectées :

$$\begin{aligned} |I_S| &= |M_i| + |M_o| \\ |I_T| &= |M_n| + |M_o| \\ |M_o| &= |M_s| + |M_r| \end{aligned}$$

Le tableau 7.2 illustre l'absence d'implémentation directe de `Drawing` dans `StandardDrawing` : les deux tiers de l'implémentation de `Drawing` sont hérités. Par contre, sur les quinze méthodes directement implémentées par `StandardDrawing`, une large majorité concerne effectivement `Drawing`.

Tableau 7.2 – Couverture de l'interface `Drawing` par la classe `StandardDrawing`.

Catégorie	`Drawing` (39) > `StandardDrawing` (15)
M_i	28 (65%) – méthodes héritées par un autre chemin
M_n	4 (9%)
M_o	11 (26%) – méthodes implémentées
M_t	43 (100%) – total

Le tableau 7.3 confirme notre première intuition. 80% des méthodes de `Drawing` peuvent être « fournies » par `CompositeFigure` (32 méthodes sur 39). Ce nombre couvre le nombre de

méthodes requises par `StandardDrawing` dans le tableau 7.2 (méthodes héritées par un autre chemin). Nous remarquons enfin que les deux tiers des méthodes définies par `CompositeFigure` correspondent à des méthodes de `Drawing` : ce nombre élevé indique une forte corrélation et, par opposition, pose la question de la corrélation entre `CompositeFigure` et `Figure`.

Tableau 7.3 – Couverture de l'interface `Drawing` par la classe `CompositeFigure`.

Catégorie	`Drawing` (39) > `CompositeFigure` (41)
M_i	7 (14%) – méthodes non fournies
M_n	9 (19%)
M_o	32 (67%) – méthodes fournies
M_t	48 (100%)

Le tableau 7.4 illustre l'héritage de `CompositeFigure` par `StandardDrawing`. Ceci ne fait que confirmer les deux tableaux précédents. On peut cependant remarquer que `StandardDrawing` remplace des méthodes de `CompositeFigure` mais ne spécialise pas. Dans cette analyse nous ne comparons que les interfaces brutes des deux classes, sans prendre en compte les interfaces héritées de la hiérarchie de `Figure`. Ceci ne change cependant pas nos conclusions (`StandardDrawing` hérite de 64 méthodes au lieu de 37 ; 7 méthodes sont des remplacements au lieu de 4).

Tableau 7.4 – Couverture de la classe `CompositeFigure` par la classe `StandardDrawing`.

Catégorie	`CompositeFigure` (41) > `StandardDrawing` (15)
M_i	37 (71%) – méthodes héritées
M_n	11 (21%) – méthodes nouvelles
M_s	0 (0%) – méthodes spécialisées
M_r	4 (8%) – méthodes remplacées
M_t	52 (100%)

Les deux analyses suivantes aident à comprendre la (faible) corrélation entre `Figure` d'une part, `StandardDrawing` et `CompositeFigure` d'autre part.

Le tableau 7.5 montre la comparaison de `CompositeFigure` avec la hiérarchie de l'interface `Figure`, c'est-à-dire l'ensemble des types impliqués entre `CompositeFigure` et `Figure`. Cette hiérarchie inclut donc l'interface `Figure` mais aussi la classe abstraite `AbstractFigure` qui, comme son nom l'indique, définit les méthodes génériques de `Figure`.

Un point notable est que `CompositeFigure` définit autant de nouvelles méthodes qu'elle en hérite. Ces nouvelles méthodes occupent même 80% de l'ensemble des définitions de `CompositeFigure`. Les 20% restant sont donc consacrées au remplacement et à la spécialisation des méthodes de `Figure` et `AbstractFigure`. Bien que `CompositeFigure` définisse bien la récursion des méthodes partagées avec `Figure`, nous concluons par le fait que sa préoccupation majeure est l'implémentation de méthodes correspondant à `Drawing` (tableau 7.3).

Tableau 7.5 – Couverture de la hiérarchie `Figure+` par la classe `CompositeFigure`.

Catégorie	`Figure+` (38) > `CompositeFigure` (41)
M_i	30 (42%)
M_n	33 (46%)
M_s	4 (6%)
M_r	4 (6%)
M_t	71 (100%)

Le tableau 7.6 compare les interfaces `Figure` et `Drawing`. Cette comparaison montre que l'ensemble des méthodes communes aux deux interfaces est réduit à cinq méthodes.

Tableau 7.6 – Couverture de l'interface `Figure` par l'interface `Drawing`.

Catégorie	`Figure` (34) > `Drawing` (39)
M_i	29 (43%)
M_n	34 (50%)
M_o	5 (7%) – méthodes partagées
M_t	68 (100%)

Ces deux dernières comparaisons permettent de juger d'une corrélation relativement faible entre `Figure` et l'implémentation du motif Composite autour de `Drawing`. Mais elles ne permettent pas d'assurer qu'une instance de `StandardDrawing` n'est jamais utilisée en tant que `Figure`. Pour cela nous devons vérifier les types des références aux instances.

- Il n'existe pas directement de variables de type `Figure` référençant une instance de `StandardDrawing`. Un traçage des appels au constructeur de `StandardDrawing` permet de vérifier ce point.
- Une analyse approfondie du code est encore nécessaire pour vérifier qu'aucun transtypage de `Drawing` vers `Figure` n'est déclarée.

Nous pouvons maintenant confirmer que l'héritage par `StandardDrawing` de la hiérarchie `Figure`, via `CompositeFigure`, est inutile : cette relation complexifie inutilement la classe `StandardDrawing` et la rend d'autant plus fragile que la corrélation est faible. Ceci et la démonstration que la majorité de l'implémentation de `CompositeFigure` est consacrée à l'interface `Drawing` aboutit aux deux décisions suivantes.

- Extraction des classes `StandardDrawing` et `CompositeFigure` de la hiérarchie `Figure` ;
- Transformation de `CompositeFigure` en aspect structurel pour être réutilisé dans les classes nécessaires, `StandardDrawing` et `GroupFigure`.

7.3.2.3 Conception

Bien que les liens de `CompositeFigure` à `Figure` soient plus faibles que ceux à `Drawing` et `StandardDrawing`, ils ne sont pas absents. En particulier la redéfinition récursive des primitives de `Figure` fait partie des usages du motif Composite. La tâche de restructuration de `CompositeFigure` est donc complexe et a des conséquences aussi bien sur `StandardDrawing`

que sur sa hiérarchie `Figure`. Ce processus est par ailleurs influencé par les aspectisations précédentes. Nous avons donc commencé par une phase de prétraitement.

- Élimination des méthodes liées aux motifs aspectisés (OBSERVER et TEMPLATE METHOD associés) ou dépréciées (*deprecated*).
- Identification dans l'interface `Figure` des méthodes liées uniquement au motif COMPOSITE.
- Utilisation des accesseurs pour les accès aux attributs hérités.
- Identification dans `CompositeFigure` de préoccupations spécifiques à `Drawing`.

Une analyse plus fine de la classe `CompositeFigure` montre en effet qu'elle factorise du code utilisé uniquement dans le contexte de `Drawing`. Suivant notre principe de modularisation, nous avons isolé celles-ci soit dans `StandardDrawing`, soit dans un aspect. Le but est ici de ne pas polluer `GroupFigure` avec ces préoccupations concernant la structure et le comportement de `Drawing`.

Un résultat fondamental de notre démarche est la distinction, grâce au critère d'adaptation (section 7.2.5) entre deux types de comportement parmi les méthodes récursives de `CompositeFigure`. Certains comportements sont spécifiques à chaque sous-type de `Figure` : nous utilisons alors le polymorphisme d'héritage et gardons des méthodes récursives. À l'inverse d'autres comportements récursifs sont génériques vis-à-vis des sous-types de `Figure` : nous utilisons alors le polymorphisme ad hoc du motif VISITOR pour définir ces comportements dans un aspect.

Bilan. Le premier objectif de cette restructuration est de transformer `CompositeFigure` en module indépendant et réutilisable à la fois par `StandardDrawing` et `GroupFigure`. Nous avons donc défini le motif COMPOSITE dans un aspect structurel `TraitCompositeFigure` composé avec le motif VISITOR générique (section 6.3.2). `TraitCompositeFigure` définit aussi la récursion des méthodes spécifiques à `Figure` et ses sous-types (pour le polymorphisme d'héritage). `GroupFigure` utilise `TraitCompositeFigure` et sous-type `Figure`. `StandardDrawing` implémente `Drawing` et utilise `TraitCompositeFigure` pour manipuler un ensemble de figures. Il bénéficie aussi de la collaboration avec VISITOR pour définir des `Visitor` sur l'ensemble des figures.

7.3.3 Autres motifs du noyau de JHotDraw

Tous les motifs du noyau (section 4.2 et tableau 4.1) n'ont pas été aspectisés. Nous en expliquons ici brièvement les raisons (toujours en nous basant sur les critères de notre démarche). Certains motifs constituent des candidats intéressants mais impraticables avec les techniques actuelles.

7.3.3.1 Mediator

Le motif MEDIATOR impliqué entre `DrawingEditor`, `Tool` et `DrawingView`, a pour intérêt de centraliser dans `DrawingEditor` la gestion des relations entre `Tool` et `DrawingView`. Ce but étant rempli, il y a peu de raisons d'aspectiser cette préoccupation. La dispersion des appels au Mediator `DrawingEditor` se fait dans des contextes propres à chaque classe.

7.3.3.2 Strategy

Les occurrences de STRATEGY utilisées par `DrawingView` remplissent par nature une seule préoccupation. `DrawingView` appelle ceux-ci en des points bien précis. Ces motifs ne présentent donc pas de symptômes intéressants pour l'aspectisation.

7.3.3.3 Adapter

Le choix d'un `Handler` dans une `Figure` se fait en fonction du contexte dynamique. La version objet du motif est en adéquation avec cette propriété : différents `Handler` peuvent être instanciés en fonction du contexte. Or il n'existe pas de version AspectJ de la variante objet ce motif : AspectJ n'est pas adéquat.

7.3.3.4 Factory Method

FACTORY METHOD est liée à l'usage du motif ADAPTER précédent. Chaque `Figure` créant les `Handler` adéquats à sa classe, FACTORY METHOD relève du polymorphisme d'héritage.

7.3.3.5 Prototype

Le motif PROTOTYPE peut être défini par un aspect structurel pour être réutilisé. Son usage reste cependant limité dans le cadre de JHotDraw.

7.3.3.6 State

L'imbrication de plusieurs occurrences du motif STATE (section 4.2.3.7) complique la compréhension du flot de contrôle des objets successifs. La substitution par des aspects, dans un cadre dynamique et événementiel, pourrait simplifier ce flot de contrôle. Cependant le principe et la faisabilité avec AspectJ restent à prouver.

7.3.3.7 Decorator

Comme montré en section 5.3.1, le motif DECORATOR aspectisé est intéressant du point de vue implémentation. Mais nous avons déjà fait remarqué que ses propriétés sont incompatibles avec le contexte de JHotDraw, qui demande la composition récursive. Ceci est encore un exemple où AspectJ n'est pas adéquat.

7.3.3.8 Memento

Une préoccupation (et une fonctionnalité) optionnelle mais intéressante est la gestion d'un historique des modifications en particulier grâce au patron MEMENTO. Il est facile de constater (par comparaison entre les versions 5.2 et 5.3 de JHotDraw, où cette fonctionnalité fut introduite) que cette implémentation est particulièrement invasive et entraîne une profonde restructuration du code. Par ailleurs, son usage implique la dispersion en plusieurs endroits d'un modèle (*template*) de classe. Si l'aspectisation est intéressante, elle est aussi peu évidente en raison du polymorphisme d'héritage du rôle `Memento`.

7.3.4 Bilan

L'application de notre méthode sur le noyau de JHotDraw permet la modularisation de plusieurs préoccupations supportées par des motifs Observer et Composite. La démarche diminue l'impact de la forte densité des motifs (section 4.3) en isolant les préoccupations avant de les recomposer avec les mécanismes d'AspectJ.

Cependant notre démarche se veut aussi pragmatique. Nous n'avons procédé à une aspectisation que si celle-ci présentait un avantage pour l'implémentation. Les résultats de la section 7.3.3 montrent de nombreuses occurrences ignorées avec notre méthode. Certains ne présentent tout simplement pas de syndrômes intéressants pour les aspects. Cependant, pour une partie d'entre eux (State, Decorator, Memento), les mécanismes d'AspectJ ne sont pas adéquats pour obtenir une transformation bénéfique.

7.4 Bilan

Nous développons une méthode de programmation pour AspectJ basée sur la complémentarité des objets et des aspects. Cette méthode vise en particulier la restructuration par les aspects. Nous proposons des critères permettant de sélectionner, décider et concevoir la modularisation d'une préoccupation dans un aspect. Deux critères en particulier sont caractéristiques d'AspectJ : le choix entre aspect comportemental (coupe-action) et aspect structurel (déclarations intertypes) ; l'adaptation (fonction de ce choix) par polymorphisme ad hoc ou d'héritage. Nous exposons des techniques pour exposer la dispersion du code ainsi qu'une méthode d'analyse de la couverture des interfaces pour la réutilisation.

Nous utilisons cette méthode pour restructurer JHotDraw et diminuer l'impact de la forte densité des motifs dans le noyau. Cette restructuration aboutit à la modularisation de plusieurs occurrences Observer et à la définition d'un motif Composite indépendant et réutilisable.

Le gain sur JHotDraw se traduit par une simplification du code des classes et la modularisation de plusieurs préoccupations. Notre démarche facilite la maintenance du cadriciel : un programmeur (chevronné) peut comprendre le fonctionnement de préoccupations sans faire d'aller-retour dans les classes ; plusieurs motifs prennent en charge l'adaptation ; les aspects (dé)tissables garantissent une meilleure évolution en cas d'imprévu.

Partie III

Conclusion et annexes

Chapitre 8

Conclusion

Sommaire

8.1 Bilan des contributions

À l'heure du bilan, nous rappelons les contributions de notre thèse ainsi que leur articulation avec nos travaux sur AspectJ et JHotDraw.

8.1.1 Densité des motifs de conception

Face à la disparité et au caractère informel des motifs, nous avons réalisé une étude de cas sur le cadriciel JHotDraw pour montrer concrètement ce qu'était une forte densité de motifs, en donner une interprétation par rapport à l'application et analyser ses conséquences sur l'implémentation du programme.

8.1.1.1 Interprétation de la densité des motifs

Une forte densité de motifs de conception indique que de nombreuses préoccupations du cadriciel sont supportées par des motifs. Nous avons distingué deux usages caractéristiques : les motifs assurant une collaboration ou une structure interne au cadriciel et les motifs permettant une extension du cadriciel.

Nous avons aussi présenté deux métriques simples de densité destinées à : évaluer la maturité du code (d'après la part assurée par les motifs) ; identifier les classes essentielles pour l'extension et les collaborations, appelées *entités**.

Ces descriptions et métriques produisent une vision plus abstraite de l'implémentation tout en restant connectées au code. Nous avons montré comment exploiter les motifs présents pour

expliquer de façon synthétique l'implémentation. Cette documentation n'est pas exhaustive mais illustre les synergies entre classes.

L'étude menée sur le cadriciel JHotDraw apporte la démonstration du pouvoir d'abstraction des motifs pour la compréhension et la documentation des programmes.

8.1.1.2 Impact des motifs sur l'implémentation objet

Pour comprendre l'impact des motifs en forte densité sur les implémentations des programmes objets, nous avons rajouté au cadriciel de nouvelles extensions supportées par des motifs. Les modifications apportées sont invasives dans le code d'abord en raison de la non-modularité des solutions objets. Mais, dans le cadre d'une forte densité, nous avons aussi constaté un impact transversal : les choix concernant un motif défini dans une première classe affectent l'implémentation d'un autre motif dans une seconde classe.

Nous avons remarqué qu'une forte densité est source d'opportunités pour la simplification et la réutilisation du code des motifs. Mais ces opportunités impliquent un compromis entre les motifs concernés et fragilisent les implémentations.

Cette étude concrète dévoile donc les divers effets d'une forte densité de motifs sur l'implémentation du programme et les motifs eux-mêmes.

8.1.2 Implémentation des motifs de conception avec AspectJ

L'implémentation des motifs de conception avec les aspects constitue le cœur de notre travail. En effet l'objectif que nous visons avec cette démarche est la modularisation des motifs, afin de faciliter d'une part la traçabilité des motifs et de leur impact, d'autre part d'étudier la composition des motifs.

L'implémentation des motifs avec AspectJ se fait par un processus de transformation de la solution objet : les éléments du motif sont pour certains transformés par les nouveaux mécanismes, pour d'autres capturés dans l'aspect. Deux catégories de solutions se distinguent – solutions idiomatiques et solutions aspects – pour lesquelles nous avons évalué la proportion relative au sein du catalogue du *GoF*.

8.1.2.1 Solutions idiomatiques et solutions aspects

L'approche idiomatique transforme partiellement la solution et modularise les autres éléments du motif dans un aspect : cette approche tend à reproduire dans l'aspect la solution objet initiale. Nous avons décrit cette approche avec les idiomes propres à AspectJ.

L'approche par aspects capture le problème visé par le patron dans une nouvelle solution, différente de la solution objet initiale. Nous avons présenté les deux cas les plus aboutis, Decorator et Visitor. Ces solutions ont des propriétés différentes de leurs homologues objets en terme d'implémentation (simplicité et généricité) mais aussi de comportement (utilisation, instantiation et récursivité). Certaines propriétés de comportement peuvent être émulées avec d'autres idiomes d'AspectJ que nous avons développés.

D'une manière générale, ces idiomes AspectJ concernent la liaison des aspects aux objets : il s'agit de définir une relation particulière entre objets ou encore d'associer l'état ou la portée d'un aspect à un objet.

8.1.2.2 Étendue du support des motifs du *GoF* avec AspectJ

Pour évaluer le résultat de l'aspectisation des motifs avec AspectJ, nous avons élaboré une grille de lecture des implémentations des motifs du *GoF*. Cette grille place le degré de réutilisation des solutions des motifs en regard des mécanismes AspectJ utilisés. Elle montre que plusieurs des motifs utilisant les idiomes AspectJ sont réutilisables. Les motifs non réutilisables peuvent bénéficier du support d'AspectJ pour les extensions structurelles à l'aide du mécanisme des déclarations intertypes. D'une manière générale, la modularisation des motifs par les aspects est effective.

Cependant le résultat de l'aspectisation par AspectJ change d'un motif à l'autre. La plupart des motifs peuvent être complètement définis dans des aspects. Mais certains motifs, même réutilisables, ne sont qu'en partie supportés par les aspects : le programmeur doit compléter la définition du motif en s'inspirant de la solution objet originale.

8.1.3 Composition des motifs de conception avec AspectJ

L'implémentation modulaire des motifs avec les aspects conduit à l'étude de la composition des motifs. Cette démarche pose la question de la modularité même de ces compositions. Nous avons aussi examiné les mécanismes d'AspectJ pour la composition structurelle et comportementale, en particulier pour la résolution d'interactions entre motifs.

8.1.3.1 Modularité et modalité de composition

Nous avons défini le concept de modalité pour décrire et discuter le degré de modularité attendu (soit aussi, le degré d'imbrication autorisé) dans la composition de deux motifs. En effet, implémenter une composition aboutit en général à un compromis entre modularisation et simplification du code, en fonction du langage de programmation et des spécifications du problème.

Nous avons illustré avec les motifs Composite et Visitor les compromis et les propriétés attachés à deux modalités de composition différentes. En particulier la collaboration élégante de Composite avec une solution générique de Visitor permet la réutilisation de la composition.

8.1.3.2 Mécanismes de composition et de résolution d'interactions entre motifs

Les compositions s'expriment dans le code avec les mécanismes des aspects. Dans AspectJ deux mécanismes se distinguent : les déclarations intertypes pour la composition structurelle et les coupes pour la composition comportementale.

La composition structurelle repose sur les déclarations intertypes d'AspectJ. Nous avons comparé les déclarations intertypes et les traits [Schärli *et al.*, 2003] pour en souligner la proximité. Cependant le modèle des traits offre une plus grande finesse dans la résolution des conflits grâce à ses opérateurs dédiés. En vue d'une intégration de ces opérateurs à AspectJ, nous avons décrit deux solutions pour le typage statique des traits.

La composition comportementale peut s'opérer avec le langage de coupe des aspects. Nous avons montré la résolution d'interactions dans une composition de motifs à l'aide des coupes sur le flot de contrôle. Cet exemple illustre l'affranchissement nécessaire – et possible avec les aspects – pour résoudre ces interactions hors du contexte des classes.

8.1.4 Méthode de programmation avec AspectJ

L'implémentation et la composition des motifs avec le langage AspectJ utilisent objets et aspects. Nous avons développé une méthode de programmation pour un usage conjoint et complémentaire des classes et des aspects. Nous préconisons à travers cette méthode une approche mesurée de l'aspectisation des motifs plutôt qu'une transformation systématique.

Cette méthode s'applique en particulier à la restructuration des objets par les aspects. Elle propose plusieurs critères permettant de décider si la modularisation par AspectJ est intéressante. Nous avons mis l'accent sur deux points (résumés dans le tableau ci-dessous) discriminant l'usage des classes et des aspects. D'une part nous distinguons les catégories de module (classe, trait et aspect) suivant différents « rôles » : les classes pour la définition d'instances et l'héritage, les traits (ou aspects structurels) pour la réutilisation de code, les aspects comportementaux pour les collaborations externes aux classes. D'autre part nous distinguons suivant le type d'adaptation offert par ces modules : les classes (et par extension les traits ou aspects structurels) supportent le polymorphisme d'héritage (redéfinition), les aspects comportementaux d'AspectJ supportent le polymorphisme ad hoc (surcharge).

Domaine	Module	Polymorphisme
Structurel	classe/trait	héritage
Comportemental	aspect	ad hoc

8.1.5 Conclusion

Nos travaux confortent l'idée que la densité des motifs de conception est un vecteur du passage à l'échelle en programmation par objets. En particulier une forte densité de motifs facilite l'abstraction du code mais fragilise l'implémentation et la modularité du programme.

Nos travaux valident aussi l'approche visant à implémenter les motifs avec les aspects. Comme indiqué par Hannemann *et al.*, la modularisation des motifs par les aspects est effective. Elle autorise le raisonnement sur l'implémentation et la composition des motifs. Les aspects permettent de s'affranchir de la classe comme module.

Enfin notre thèse explore les bons usages de la programmation par aspects. Les résultats obtenus avec AspectJ sont inégaux en terme de simplicité d'implémentation, de réutilisation et de propriétés des motifs. Les motifs aspectisés, les interactions comportementales, la collaboration de Composite et Visitor indiquent l'usage bénéfique des coupes pour définir une liaison tardive dans les implémentations et compositions. Les idiomes nécessaires à AspectJ pour la liaison aux objets ou encore la meilleure capacité des traits pour la résolution des conflits structurels soulignent certaines lacunes du langage. C'est pourquoi nous préconisons en pratique une utilisation mesurée d'AspectJ.

8.2 Travaux futurs et perspectives

Les résultats de notre thèse nous inspirent plusieurs pistes de recherche tant dans l'immédiat qu'à plus long terme. Nous exposons celles-ci selon trois directions principales.

8.2.1 Sur l'étude des motifs et de la densité

Cette thèse examine par une analyse qualitative les effets de la densité des motifs et les problèmes liés à leur implémentation et à leur composition dans le contexte de JHotDraw. Elle nous a permis d'illustrer par des exemples concrets notre intuition : la densité des motifs est un vecteur du passage à l'échelle en programmation par objets.

Cependant pour confirmer cette intuition, il nous faut envisager une étude plus large sur différents codes sources. Cette étude demande une nouvelle méthode d'analyse. En effet pour une tâche de cette ampleur, nous ne pouvons pas effectuer un examen visuel du code, au cas par cas. Or, étant donné l'absence de traçabilité des implémentations des motifs dans le code objet, nous envisageons l'usage des outils de détection des motifs [Guéhéneuc, 2003] : ces outils permettent à l'heure actuelle d'identifier les motifs candidats en présence. Dans un second temps, nous envisageons d'améliorer ces outils pour détecter et tracer précisément le code d'un motif malgré la dispersion et le mélange. Cette analyse doit aussi s'appuyer sur la rétroconception et la restructuration d'un code affecté par les implémentations des motifs, tel que nous l'avons effectué au chapitre 4.

Cette thèse ne propose pas de modèle précis de la composition des motifs, bien que nous ayons recherché ce résultat pendant longtemps. Par exemple les modalités s'approchent de l'idée d'un modèle certes universel, mais sont informels et trop abstrait pour pouvoir manipuler du code. Ceci tient en particulier à la disparité des patrons et à la relativité des motifs, qui gênent leur classification et leur formalisation (chapitre 2 et problématique page 48). En fait une étape préliminaire à la définition d'un modèle de composition est la question de la composition *dans quel but ?* Nous nous sommes intéressés à la composition des motifs du point de vue de l'implémentation et de la modularité.

D'autres modèles sont à étudier, pour la conception de compositions ou la prédiction des interactions par exemple. Un modèle qui nous semble potentiellement intéressant, pour la simplification et la généralisation qu'il offre, est celui des *Meta Patterns* [Pree, 1994]. Ce modèle est focalisé sur les relations et extensions communes à de nombreux motifs. Cependant la question du *quoi modéliser* reste aussi ouverte, tant la structure statique d'un motif peut être différente de sa structure dynamique [Gamma *et al.*, Introduction, pages 22–23]. En poussant ce raisonnement, [Ziane, 2004] propose ainsi de modéliser les problèmes des patrons par des contraintes de découplage et de qualité, plutôt que les motifs eux-mêmes : étudier la composition sous cet angle ouvre sans doute la voie à la prédiction des effets des compositions de motifs.

Enfin la relation entre densité des motifs et qualité du code reste à affiner. Nous proposons d'analyser en parallèle les métriques classiques de qualité et de densité pour déterminer si cette relation existe. À terme, nous espérons le développement de métriques de densité intégrées aux métriques de qualité.

Traçabilité des choix de conception

Un des thèmes sous-jacents à notre thèse et qui nous semble particulièrement important à approfondir est la traçabilité des choix de conception. Ce thème est évidemment associé à la traçabilité des motifs et en particulier inspiré par les effets de la densité des motifs : nous avons montré dans l'implémentation objet de JHotDraw l'impact transversal des motifs sur d'autres motifs, qui aboutissait à des choix de conception et des transformations de code conséquentes. Ce thème recouvre de larges domaines de recherche comme la conception et la modélisation, le versionnage, les mécanismes d'évolution et les techniques de restructuration.

8.2.2 Sur l'implémentation et la composition des motifs avec les aspects

Une des principales critiques que l'on peut faire sur notre étude des motifs est d'avoir restreint le cadre à Java et AspectJ. Le choix de ces langages (en particulier pour AspectJ) était motivé par la maturité atteinte et les études déjà existantes. Pourtant nous nous sommes heurtés pendant nos travaux à des problèmes dus avant tout à des choix de conception (pour ne pas dire des défauts) de ces langages. Ceci peut nous interpeller sur la portée de nos généralisations à partir d'AspectJ, ainsi que sur des usages impossibles avec AspectJ mais possibles avec d'autres langages d'aspects. En particulier les idiomes d'AspectJ pour la liaison aux objets nous semblent spécifiques à ce langage. Nous avons par exemple signalé des solutions alternatives du motif Observer avec d'autres technologies aspects comme Caesar [Mezini et Ostermann, 2003] et Reflex [Tanter, 2004]. Une étude extensive des motifs à la manière de [Hannemann et Kiczales, 2002] avec différents langages d'aspect serait particulièrement intéressante (même s'il s'agit d'un travail fastidieux). L'expérience aidant, un catalogue des compositions de motifs et de leurs propriétés pourra être construit.

À l'issue de cette thèse, notre conviction est que les aspects peuvent améliorer la composition des motifs et que, réciproquement, l'étude de la composition des motifs peut fournir des pistes pour de nouveaux mécanismes des langages d'aspects (ou l'amélioration de mécanismes existant). Ceci se base en particulier sur notre expérience avec les interactions comportementales en AspectJ, où les coupes sur le flot de contrôle permettent de dépasser le cadre de la classe dominante dans la résolution. Cependant ces travaux reposent aussi sur l'expérience acquise en composition de motifs dans les programmes purement objets : l'étude quantitative que nous envisageons (section 8.2.1) peut donc servir de base à l'amélioration des langages d'aspects.

Le projet AJHotDraw est un projet similaire au notre par son principe de restructuration de JHotDraw avec AspectJ[7]. Cependant l'objectif et les pistes de recherche sont différentes puisqu'il s'agit d'expérimenter les techniques de détection d'aspects potentiels (*aspect mining*) [Marin *et al.*, 2007]. Ces travaux ne ciblent donc pas spécifiquement les motifs de conception et encore moins leur composition. Il est cependant évident qu'il existe des convergences entre ces deux projets, qui restent à rapprocher.

8.2.3 Sur AspectJ

Au fil de cette thèse nous avons exploré et exposé différents mécanismes d'AspectJ et souligné leurs défauts. En premier lieu viennent les idiomes d'AspectJ liés aux liaisons avec les objets : le modèle d'instantiation-portée d'AspectJ, bien qu'il ait l'avantage de simplifier l'usage du langage et de limiter les risques d'erreur à l'exécution, semble être mis en cause. Nous n'avons cependant pour l'instant pas de solution à ce défaut. [Pearce et Noble, 2006] montre comment expliciter les relations entre objets avec des bibliothèques d'aspects AspectJ : ces relations étant à la base de nombreux motifs, il serait intéressant de rééditer cette étude avec ces bibliothèques.

La comparaison des déclarations intertypes avec le modèle des traits est en faveur de ce dernier. L'intégration des opérateurs de composition dans les déclarations intertypes est selon nous à envisager. Nous avons montré deux approches possibles pour cette intégration en accord avec le typage statique de Java : un typage nominatif dans la ligne courante de la spécification du langage Java et un typage paramétrique plus sophistiqué.

[7] Page d'accueil `http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw`.

Bien qu'il soit possible de spécialiser des coupes abstraites et de réutiliser des coupes concrètes définies dans un autre aspect, il n'est pas possible avec AspectJ de spécialiser des coupes concrètes. Un tel mécanisme marquerait le pas vers un polymorphisme d'héritage classique dans les aspects au lieu du seul polymorphisme ad hoc actuel. Cependant ce mécanisme implique un changement de technique de tissage, qui doit être dynamique (pour tisser et détisser les coupes) à la façon de certaines technologies aspects actuelles (Steamloom [Bockisch *et al.*, 2004], JAsCo [Suvée *et al.*, 2003]).

Un problème concret auquel nous avons été confronté avec AspectJ est la prise en compte de l'état d'un objet dans le langage de coupe : il est délicat de vérifier si l'objet est dans un état consistant ou inconsistant sans violer la modularité. [Kiczales et Mezini, 2005] suggère l'usage des *Typestates* comme moyen d'abstraction des états des objets [DeLine et Fähndrich, 2004].

L'exemple de la solution générique VISITOR en AspectJ illustre une autre piste d'exploration pour l'implémentation des langages d'aspects et des tisseurs en particulier. La forme de la solution aspectisée est proche de la forme des multi-méthodes, que le patron VISITOR cherche justement à émuler. Cette proximité indique que les nombreux travaux sur l'implémentation des multi-méthodes peuvent à leur tour servir pour implémenter et optimiser le tissage des aspects.

Bien que ces changements concernent en priorité AspectJ, nous pouvons les envisager dans d'autres langages, en particulier dans des cadriciels ou langages compatibles comme Reflex et abc [Avgustinov *et al.*, 2005].

Bibliographie

[Agerbo et Cornils, 1998] cité pages 14, 17, 25
Ellen Agerbo and Aino Cornils.
How to Preserve the Benefits of Design Patterns.
In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 134–143. ACM Press, 1998. ISBN: 1-58113-005-8.

[Akşit, 2003] cité pages 155, 157, 158
Mehmet Akşit, editor.
Proceedings of the 2nd Conference on Aspect-Oriented Software Development (AOSD'03). ACM Press, 2003. ISBN: 1-58113-660-9.

[Albin-Amiot, 2003] cité pages 28
Hervé Albin-Amiot.
Idiomes et patterns Java: application à la synthèse de code et à la détection.
Thèse de doctorat, École des Mines de Nantes et Université de Nantes, 2003.

[Aldrich, 2005] cité pages 40
Jonathan Aldrich.
Open Modules: Modular Reasoning About Advice.
In Black [2005], pages 144–168. ISBN: 3-540-27992-X.

[Arnout, 2004] cité pages 25
Karine Arnout.
From Patterns to Components.
Ph.D. thesis, Swiss Institute of Technology, Zurich (ETH Zurich), 2004.

[Avgustinov *et al.*, 2005] cité pages 149
Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble.
abc: an Extensible AspectJ Compiler.
In Mezini and Tarr [2005], pages 87–98. ISBN: 1-59593-042-6.

[Baroni *et al.*, 2003] cité pages 48
Aline Lúcia Baroni, Yann-Gaël Guéhéneuc, and Hervé Albin-Amiot.
Design Patterns Formalization.
Technical report 03/03/INFO, Computer Science Department, École des Mines de Nantes, 2003.

[Bergmans et Akşit, 2001] cité pages 42
Lodewijk Bergmans and Mehmet Akşit.
Composing Crosscutting Concerns Using Composition Filters.
Communications of the ACM, 44(10):51–57, 2001.

[Black, 2005] cité pages 151, 157
Andrew Black, editor.
Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05), volume 3586 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. ISBN: 3-540-27992-X.

[Bockisch *et al.*, 2004] cité pages 149
Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann.
Virtual Machine Support for Dynamic Join Points.
In Gail Murphy and Karl Lieberherr, editors, *Proceedings of the 3rd Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 83–92. ACM Press, 2004. ISBN: 1-58113-842-3.

[Bosch et Mitchell, 1998] cité pages 154
Jan Bosch and Stuart Mitchell, editors.
Object-Oriented Technology, ECOOP'97 Workshop Reader, volume 1357 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. ISBN: 3-540-64039-8.

[Bosch, 1998] cité pages 18, 23, 24, 26
Jan Bosch.
Design Patterns as Language Constructs.
Journal of Object-Oriented Programming (JOOP), 11(2):18–32, 1998.

[Braux et Noyé, 1999] cité pages 23
Mathias Braux et Jacques Noyé.
Changement dynamique de comportement par composition de schémas de conception.
Jacques Malenfant et Roger Rousseau, éditeurs, *Langages et modèles à objets (LMO'99)*, pages 147–162. Hermès, 1999.

[Cacho *et al.*, 2006] cité pages 44, 102
Nélio Cacho, Cláudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thaís Vasconcelos Batista, and Carlos José Pereira de Lucena.
Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming.
In Filman [2006], pages 109–121. ISBN: 1-59593-300-X.

[Chambers *et al.*, 2000] cité pages 25
Craig Chambers, Bill Harrison, and John Vlissides.
A Debate on Language and Tool Support for Design Patterns.
In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL'00)*, pages 277–289. ACM Press, 2000. ISBN: 1-58113-125-9.

[Clarke et Walker, 2001] cité pages 43
Siobhán Clarke and Robert Walker.
Composition Patterns: An Approach to Designing Reusable Aspects.
In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 5–14. IEEE Computer Society Press, 2001. ISBN: 0-7695-1050-7.

[Cointe *et al.*, 2005] cité pages 33
Pierre Cointe, Hervé Albin-Amiot, and Simon Denier.
From (Meta) Objects to Aspects.

In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem de Roever, editors, *Proceedings of the 3rd International Symposium on Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 70–94. Springer-Verlag, 2005.

[Coplien et Shmidt, 1995] cité pages 158, 159
James Coplien and Douglas Shmidt, editors.
Pattern Languages of Program Design.
Addison-Wesley, 1995. ISBN: 0-201-60734-4.

[De Volder et D'Hondt, 1999] cité pages 41
Kris De Volder and Theo D'Hondt.
Aspect-Oriented Logic Meta Programming.
In Pierre Cointe, editor, *Proceedings of the 2nd Conference on Meta-Level Architectures and Reflection (Reflection'99)*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer-Verlag, 1999. ISBN: 3-540-66280-4.

[DeLine et Fähndrich, 2004] cité pages 149
Robert DeLine and Manuel Fähndrich.
Typestates for Objects.
In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer-Verlag, 2004. ISBN: 3-540-22159-X.

[Denier et Cointe, 2006a] cité pages x, xi, 6, 7
Simon Denier and Pierre Cointe.
Expression and Composition of Design Patterns with AspectJ.
RSTI - L'objet, 12(2–3):41–61. Hermès-Lavoisier, 2006.

[Denier et Cointe, 2006b] cité pages x, 6
Simon Denier and Pierre Cointe.
Understanding Design Patterns Density with Aspects: A Case Study in JHotDraw using AspectJ.
In Welf Löwe and Mario Südholt, editors, *Proceedings of the International Workshop on Software Composition (SC'06)*, volume 4089 of *Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, 2006.

[Denier, 2005] cité pages xi, 8
Simon Denier.
Traits Programming with AspectJ.
In Pierre Cointe and Mario Südholt, editors, *RSTI - L'objet*, 11(3):69–86. Hermès-Lavoisier, 2005.

[Dijkstra, 1974] cité pages 31
Edsger Dijkstra.
On the Role of Scientific Thought.
`http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html`, 1974.

[Douence et Teboul, 2004] cité pages 116, 117, 118
Rémi Douence and Luc Teboul.
A Pointcut Language for Control-Flow.

In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd Conference on Generative Programming and Component Engineering (GPCE'04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114. Springer-Verlag, 2004. ISBN: 3-540-23580-9.

[Ducasse, 1997] cité pages 26, 29
Stéphane Ducasse.
Message Passing Abstractions as Elementary Bricks for Design Pattern Implementation.
In Bosch and Mitchell [1998], pages 96–99. ISBN: 3-540-64039-8.

[Filman *et al.*, 2005] cité pages 3, 36, 156
Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors.
Aspect-Oriented Software Development.
Addison-Wesley, Boston, 2005. ISBN: 0-321-21976-7.

[Filman et Friedman, 2000] cité pages 40
Robert Filman and Daniel Friedman.
Aspect-Oriented Programming is Quantification and Obliviousness.
In *Workshop on Advanced Separation of Concerns (OOPSLA'00)*, 2000.

[Filman, 2006] cité pages 152, 157
Robert Filman, editor.
Proceedings of the 5th Conference on Aspect-Oriented Software Development (AOSD'06). ACM Press, 2006. ISBN: 1-59593-300-X.

[Fowler *et al.*, 1999] cité pages 28
Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[Gamma *et al.*, 1994] cité pages xi, 2, 5, 13, 14, 15, 22, 48, 147, 165
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, Massachusetts, 1994. ISBN: 0-201-63361-2.

[Gamma et Beck, 2002] cité pages 2, 22, 42, 60, 61
Erich Gamma and Kent Beck.
JUnit: A Cook's Tour.
`http://junit.sourceforge.net/doc/cookstour/cookstour.htm`, 2002.

[Garcia *et al.*, 2005] cité pages 44
Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos José Pereira de Lucena, and Arndt von Staa.
Modularizing Design Patterns with Aspects: A Quantitative Study.
In Mezini and Tarr [2005], pages 3–14. ISBN: 1-59593-042-6.

[Gil et Lorenz, 1997] cité pages 15, 16, 17
Joseph Gil and David Lorenz.
Design Patterns vs Language Design.
In Bosch and Mitchell [1998], pages 108–111. ISBN: 3-540-64039-8.

[Griswold *et al.*, 2006] cité pages 41
William Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan.
Modular Software Design with Crosscutting Interfaces.
IEEE Software, 23(1):51–60, 2006.

[Guéhéneuc, 2003] cité pages 14, 28, 147, 165
Yann-Gaël Guéhéneuc.
Un cadre pour la traçabilité des motifs de conception.
Thèse de doctorat, École des Mines de Nantes et Université de Nantes, 2003.

[Gybels et Brichau, 2003] cité pages 40
Kris Gybels and Johan Brichau.
Arranging Language Features for Pattern-Based Crosscuts.
In Akşit [2003], pages 60–69. ISBN: 1-58113-660-9.

[Hannemann et Kiczales, 2002] cité pages xi, 5, 43, 49, 76, 78, 84, 85, 87, 92, 98, 107, 148
Jan Hannemann and Gregor Kiczales.
Design Pattern Implementation in Java and AspectJ.
In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 161–173. ACM Press, 2002. ISBN: 1-58113-471-1.

[Harrison et Ossher, 1993] cité pages 3
William Harrison and Harold Ossher.
Subject-Oriented Programming (A Critique of Pure Objects).
In *Proceedings of the 8th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, pages 411–428. ACM Press, 1993.

[Isberg, 2005] cité pages 42
Wes Isberg.
Design with Pointcuts to Avoid Pattern Density.
`http://www-128.ibm.com/developerworks/java/library/j-aopwork7/index.html`, 2005.
AOP@Work Series.

[Kiczales *et al.*, 1991] cité pages 3
Gregor Kiczales, Jim des Rivières, and Daniel Bobrow.
The Art of the Metaobject Protocol.
MIT Press, 1991. ISBN: 0-262-61074-4.

[Kiczales *et al.*, 1997] cité pages 3, 32, 33
Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin.
Aspect-Oriented Programming.
In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[Kiczales *et al.*, 2001] cité pages 33
Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold.
An Overview of AspectJ.
In Jørgen Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001. ISBN: 3-540-42206-4.

[Kiczales et Mezini, 2005] cité pages 41, 149
Gregor Kiczales and Mira Mezini.
Aspect-Oriented Programming and Modular Reasoning.
In Gruia-Catalin Roman, William Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 49–58. ACM Press, 2005.

[Lieberherr *et al.*, 2001] cité pages 33, 38, 42
Karl Lieberherr, Doug Orleans, and Johan Ovlinger.
Aspect-Oriented Programming with Adaptive Methods.
Communications of the ACM, 44(10):39–41, 2001.

[Lieberherr, 1996] cité pages 3
Karl Lieberherr.
Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.
PWS Publishing Company, 1996. ISBN: 0-534-94602-X.

[Lieberman, 1986] cité pages 20
Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems.
In *Proceedings of the 1st Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPLSA'86)*, pages 214–223. ACM Press, 1986. ISBN: 0-89791-204-7.

[Lopes, 2005] cité pages 32
Cristina Lopes.
AOP: An Historical Perspective (What's in a Name?).
In Filman et al. [2005], pages 97–122. ISBN: 0-321-21976-7.

[Marin *et al.*, 2007] cité pages 148
Marius Marin, Leon Moonen, and Arie van Deursen.
An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw.
In *Proceedings of the IEEE International Conference on Source Code Analysis and Manipulation (SCAM'07)*, 2007.
To appear.

[Martin, 2000] cité pages 19
Robert Martin.
Design Principles and Design Patterns.
`http://objectmentor.com/resources/articles/Principles_and_Patterns.pdf`, 2000.

[Mezini et Ostermann, 2003] cité pages xii, 8, 43, 148
Mira Mezini and Klaus Ostermann.

Conquering Aspects with Caesar.
In Akşit [2003], pages 90–99. ISBN: 1-58113-660-9.

[Mezini et Tarr, 2005] cité pages 151, 154
Mira Mezini and Peri Tarr, editors.
Proceedings of the 4th Conference on Aspect-Oriented Software Development (AOSD'05). ACM Press, 2005. ISBN: 1-59593-042-6.

[Minjat *et al.*, 2005] cité pages xii, 8
Florian Minjat, Alexandre Bergel, Pierre Cointe et Stéphane Ducasse.
Mise en symbiose des traits et des classboxes, application à l'expression des collaborations.
Langages et Modèles à Objets, pages 33–46. Hermès-Lavoisier, 2005.

[Nordberg III, 2001a] cité pages 41
Martin Nordberg III.
Aspect-Oriented Dependency Inversion.
In *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems – OOPSLA'01*, 2001.

[Nordberg III, 2001b] cité pages 41
Martin Nordberg III.
Aspect-Oriented Indirection – Beyond Object-Oriented Design Patterns.
In *Proceedings of the Workshop "Beyond Design: Patterns (mis)used" – OOPSLA'01*, 2001.

[Norvig, 1998] cité pages 25
Peter Norvig.
Design Patterns in Dynamic Programming.
`http://norvig.com/design-patterns/`, 1998.

[Ossher et Tarr, 2000] cité pages 32
Harold Ossher and Peri Tarr.
Multi-Dimensional Separation of Concerns and The Hyperspace Approach.
In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Ostermann *et al.*, 2005] cité pages 41
Klaus Ostermann, Mira Mezini, and Christoph Bockisch.
Expressive Pointcuts for Increased Modularity.
In Black [2005], pages 214–240. ISBN: 3-540-27992-X.

[Parnas, 1972] cité pages 40, 124
David Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM, 15(12):1053–1058, 1972.

[Pearce et Noble, 2006] cité pages 148
David Pearce and James Noble.
Relationship Aspects.
In Filman [2006], pages 75–86. ISBN: 1-59593-300-X.

[Pree, 1994] cité pages 147
Wolfgang Pree.
Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design.
In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 150–162. Springer-Verlag, 1994. ISBN: 3-540-58202-9.

[Reenskaug, 1979] cité pages 53
Trygve Reenskaug.
MVC: Xeroc PARC 1978–79.
`http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`, 1979.

[Riehle, 1997] cité pages 23, 24, 72
Dirk Riehle.
Composite Design Patterns.
In *Proceedings of the 12th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, pages 218–228. ACM Press, 1997.

[Schärli *et al.*, 2003] cité pages xi, 8, 39, 100, 109, 145
Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black.
Traits: Composable Units of Behavior.
In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, 2003. ISBN: 3-540-40531-3.

[Smith, 1984] cité pages 3
Brian Smith.
Reflection and Semantics in LISP.
In *Proceedings of the 11th Symposium on Principles of Programming Languages (POPL'84)*, pages 23–35. ACM Press, 1984. ISBN: 0-89791-125-3.

[Soukup, 1995] cité pages 18, 23, 27
Jiri Soukup.
Implementing Patterns.
In Coplien and Shmidt [1995], pages 395–412. ISBN: 0-201-60734-4.

[Suvée *et al.*, 2003] cité pages 149
Davy Suvée, Wim Vanderperren, and Viviane Jonckers.
JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development.
In Akşit [2003], pages 21–29. ISBN: 1-58113-660-9.

[Tanter *et al.*, 2003] cité pages 43
Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe.
Partial Behavioral Reflection: Spatial and Temporal Selection of Reification.
In Ron Crocker and Guy Steele Jr., editors, *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 27–46. ACM Press, 2003.

[Tanter, 2004] cité pages xii, 8, 43, 148
Éric Tanter.

From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming.
Ph.D. thesis, École des Mines de Nantes, Université de Nantes, and University of Chile, 2004.

[Tatsubori et Chiba, 1998] cité pages 27
Michiaki Tatsubori and Shigeru Chiba.
Programming Support of Design Patterns with Compile-time Reflection.
In *Proceedings of the Workshop on Reflective Programming in C++ and Java – OOPSLA'98*, pages 56–60, 1998.

[Yacoub et Ammar, 2003] cité pages 23
Sherif Yacoub and Hany Ammar.
Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems, chapter 3.
Addison-Wesley, 2003. ISBN: 0-201-77640-5.

[Ziane, 2004] cité pages 28, 147
Mikal Ziane.
Transformations de programmes et de requêtes pour automatiser ou faciliter la production et la maintenance du logiciel.
Habilitation à diriger des recherches, Université Paris 6, 2004.

[Zimmer, 1995] cité pages 16, 22
Walter Zimmer.
Relationships between Design Patterns.
In Coplien and Shmidt [1995], pages 345–364. ISBN: 0-201-60734-4.

Listes

Définitions

Illustrations

Tableaux

Sources

Exemples

Glossaire

A

action

En anglais : *advice*.

Une action correspond à l'association d'une coupe avec un bloc d'instructions. Elle décrit la réaction de l'aspect à chaque point de jonction capturé par la coupe. Les paramètres du point de jonction exposés par la coupe ont une portée limitée au bloc de l'action. Une action est donc similaire à une méthode dont la signature serait la coupe.
Voir aussi : coupe, interface transversale, section 3.1.3.3 page 35.

aspect

Un aspect est une unité de modularisation pour préoccupation transversale. Un aspect modularise la déclaration et la définition des coupes et actions permettant l'implémentation de cette préoccupation. Suivant le langage d'aspects, un aspect peut (ne pas) être assimilé à une classe et définir des méthodes et attributs.
Voir aussi : préoccupation transversale, coupe, action, section 3.1.3.4 page 35.

C

cadriciel

En anglais : *framework*.

Un cadriciel (ou cadre d'applications) est un squelette d'application pour un domaine particulier. Un cadriciel permet la réutilisation d'un ensemble de classes collaborant entre elles. L'utilisation d'un cadriciel est basée sur l'héritage et le principe d'Hollywood : l'utilisateur spécialise les classes abstraites du cadriciel pour son application ; le flot de contrôle est défini par le cadriciel, qui appelle les classes spécialisées par l'utilisateur quand c'est nécessaire.
Voir aussi : entité.

coupe

En anglais : *pointcut*.

Une coupe est la description d'un ensemble de points de jonction à capturer dans un programme ou lors de son exécution. Le langage de coupe est associé au modèle de point de jonction pour pouvoir décrire de façon abstraite les points de jonction intéressants. Une coupe permet aussi de décrire l'extraction de paramètres dans le contexte du point de jonction et d'exposer ces paramètres.
Le terme coupe désigne aussi la capture proprement dite d'un point de jonction. Pour lever l'ambiguïté, la description est alors appelée déclaration de coupe (*pointcut designator*).
Voir aussi : interface transversale, section 3.1.3.2 page 34.

D

domaine comportemental

Le domaine comportemental d'un langage de programmation concerne les collaborations et les interactions entre éléments du langage. Les mécanismes de ce domaine permettent la sélection ou la modification du comportement en fonction du contexte. L'envoi polymorphique de message, l'accès aux attributs des objets, la coupe par les aspects, sont des mécanismes comportementaux.
Voir aussi : section 3.1.4 page 36, section 6.2.3 page 103.

domaine structurel

Le domaine structurel d'un langage de programmation concerne la déclaration et la définition des éléments dans le langage. Les mécanismes de ce domaine permettent la définition ou l'ajout d'un nouvel élément. La définition des classes et des méthodes, la composition et l'héritage de classes, la déclaration intertype pour les aspects, sont des mécanismes structurels.
Voir aussi : section 3.1.4 page 36, section 6.2.3 page 103.

E

entité

Une entité désigne un concept essentiel du métier d'une application. Chaque entité existe au niveau de la conception et se transpose dans l'implémentation. Une entité joue donc un rôle important dans la traçabilité. L'entité permet de parler d'un seul concept même si l'implémentation la décompose en plusieurs éléments : dans un cadriciel une entité est souvent representée par une interface Java et une classe (abstraite) implémentant l'interface.

I

idiome

Un idiome de programmation décrit un usage courant mais spécifique à un langage de programmation. Cet usage concerne les mécanismes du langage. Un idiome est donc d'un niveau plus concret et plus simple qu'un motif ou une solution. Une solution est souvent l'adaptation du motif aux idiomes du langage ciblé.

interface transversale

En anglais : *crosscutting interface*.

Une interface transversale désigne la frontière entre différents modules dont au moins un aspect. L'ensemble des coupes de l'aspect définit l'interface transversale de l'assemblage des modules.
Voir aussi : aspect, action, section 3.1.5 page 40.

M

modalité

La modalité d'une composition de motifs de conception exprime le degré d'imbrication (ou de dépendances) des deux motifs attendu dans la composition. C'est une abstraction conceptuelle de l'impact de la composition sur l'implémentation. Elle

permet d'évaluer si la modularité atteinte dans l'implémentation est adéquate.
Voir aussi section 6.2.1 page 101.

motif de conception

Un motif de conception identifie et décrit la solution générique au problème abordé par le patron homonyme. Le motif est donc partie intégrante de la description du patron. Il regroupe principalement les rubriques *participants*, *structure*, *collaborations* ainsi que les détails éventuellement fournis dans la rubrique *conséquences*. Un motif est abstrait, indépendant du langage d'implémentation.
Voir aussi : patron de conception, solution, occurrence.

O

occurrence

Une occurrence est la spécialisation d'un motif de conception pour un langage et une application donnée. C'est une concrétisation du motif dans le code mais aussi seulement une des formes possibles du motif. Une occurrence est donc le résultat final de l'application du patron et en particulier de la sélection des options parmi les variations du patron.
Un programme peut donc contenir plusieurs occurrences du même motif, certaines corrélées et d'autres indépendantes. Une occurrence peut être réutilisable à l'échelle du programme, c'est-à-dire pour le domaine pour lequel elle est conçue.
Voir aussi : motif de conception, solution.

P

patron de conception

En anglais : *design pattern*.

Un patron de conception identifie et décrit une solution simple et élégante à un problème récurrent de conception en programmation par objets [Gamma *et al.*, 1994]. Nous utilisons *patron de conception* pour distinguer l'ensemble {*nom, problème, solution, compromis*}, qui guide le programmeur, du *motif de conception*, qui est la solution générale résultant de ce processus.
Cette utilisation des mots *patron* et *motif* essaie de concilier la terminologie anglaise (*pattern*), la distinction nécessaire entre un processus (un *patron*) et le résultat du processus (un *motif*). D'après [Guéhéneuc, 2003].
Voir aussi : section 2.1 page 13, motif de conception.

point de jonction

En anglais : *join point*.

Un point de jonction est un élément de la sémantique d'un programme sur lequel un aspect peut effectuer une action. Les différents types de points de jonction accessibles sont définis par un modèle de point de jonction, dépendant du langage de programmation visé. Suivant ce modèle, un programme peut être vu comme un graphe de points de jonction statiques, son exécution par une trace de points de jonction dynamiques (événements).
Voir aussi : coupe, section 3.1.3.1 page 33.

préoccupation transversale

En anglais : *crosscutting concern*.

Une préoccupation est dite transversale si elle ne peut pas être modularisée correctement dans une décomposition hiérarchique du programme. Dans une telle décomposition, l'implémentation de la préoccupation se retrouve dispersée et mélangée à travers les modules des autres préoccupations.
Voir aussi : aspect, section 3.1.2 page 32.

S

solution

La solution d'un patron de conception est la spécialisation du motif attaché pour un langage de programmation donné. La solution retient donc l'aspect générique du motif mais utilise les mécanismes du langage pour implémenter ces éléments génériques. Le motif peut donc se trouver transformer par l'utilisation de mécanismes spécifiques au langage, y compris les idiomes.
Une solution correspond à un exemple du motif à imiter dans un langage donné.
Voir aussi : motif de conception, idiome, occurrence.

Index

Expression et composition des motifs de conception avec les aspects

Simon Denier

Résumé

Les patrons de conception répertorient les bonnes pratiques de la programmation par objets. Les solutions des patrons, appelées motifs, apparaissent avec une densité croissante dans les bibliothèques et cadriciels. Les effets de cette densité sur la modularité, l'adaptation et la réutilisation des programmes sont mal connus. Or la dispersion et le mélange du code lié à l'implémentation des motifs rendent difficile l'étude de ces effets. La programmation par aspects est une technique nouvelle dédiée au traitement de ces deux symptômes. En modularisant les motifs dans des aspects, nous pouvons analyser de manière plus fine les problèmes d'implémentation et de composition des motifs liés à leur densité.
Cette thèse aborde les problèmes de la densité, de l'implémentation et de la composition des motifs avec AspectJ, une extension de Java pour les aspects. À partir du cas concret du cadriciel JHotDraw, nous montrons qu'une forte densité est un facteur de risque sur la modularité et l'adaptation d'un programme objet. Nous présentons la transformation des motifs à l'aide des aspects et nous décrivons les idiomes d'AspectJ supportant leur modularisation. Nous examinons la modularité et la réutilisation des compositions de motifs définies avec les aspects. Nous proposons la résolution des interactions entre motifs à l'aide du langage de coupe des aspects. Enfin nous développons une méthode de programmation avec AspectJ basée sur l'usage conjoint des classes et des aspects. Ces travaux nous permettent de conclure sur l'intérêt des aspects comme moyen d'étude et de traitement de la densité des motifs. Ils ouvrent également des pistes pour l'amélioration des langages d'aspects.

Mots-clés : patron de conception, motif, implémentation, composition, densité, objet, aspect, langage de programmation

Abstract

Design patterns are considered as good practices of object-oriented programming. Patterns solutions, which we called motifs, appear with an increasing density in libraries and frameworks. Little is known about the effects of a high density of motifs on modularity, adaptation, and reuse of code. Studying density is difficult due to scattering and tangling of motif implementation. Aspect-oriented programming is a new technique dedicated to the treatment of code scattering and tangling. Then modularizing motifs with aspects should enable us to analyze implementation and composition of motifs related to their density.
This thesis addresses the problems of density, implementation and composition of motifs with AspectJ, an extension of Java for aspects. Based on a case study with the JHotDraw framework, we illustrate how a high density of motifs weakens modularity and adaptation of code. We present transformation of motifs with the help of aspects and we describe AspectJ idioms supporting their modularization. We inspect modularity and reuse of motifs composition defined with aspects. We demonstrate how aspects crosscutting languages help solving motifs interactions. Finally we develop a programming model for AspectJ based on the joint use of classes and aspects. This work enlightens how aspects facilitate studying and handling a high density of motifs. Moreover, it opens perspectives for the improvement of aspects languages.

Keywords: design pattern, motif, implementation, composition, density, object, aspect, programming language